INTELLIGENT MANAGEMENT OF INTER-THREAD SYNCHRONIZATION DEPENDENCIES FOR CONCURRENT PROGRAMS

by

Hyoun Kyu Cho

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2014

Doctoral Committee:

Professor Scott Mahlke, Chair
Terence Kelly, HP Labs
Professor Stéphane Lafortune
Professor Trevor Mudge
Assistant Professor Satish Narayanasamy

To my family

ACKNOWLEDGEMENTS

First, I would like to express my sincerest gratitude to my advisor, Prof. Scott Mahlke, for his continuous guidance and support. In Korean culture, teacher-student relationships are thought to be very important and respected very much. They are often metaphorically compared to father-son relationships. While I think my relationship with Scott is quite close to the ideal, it is not because I am a traditional Korean but it is because Scott is a true mentor.

I also owe thanks to the remaining members of my dissertation committee, Prof. Stéphane Lafortune, Prof. Trevor Mudge, Prof. Satish Narayanasamy, and Dr. Terence Kelly. They gave me a lot of advices, supports, ideas, and encouragements. This dissertation also includes results from joint work with Dr. Yin Wang, Dr. Hongwei Liao, Dr. Tipp Moseley, Dr. Richard Hank, and Dr. Derek Bruening. I am grateful for these wonderful teachers that I could meet over the graduate school years.

I was fortunate to be among my fellow comrades in the Compilers Creating Custom Processors (CCCP) group. They not only helped me intellectually with the discussions and feedbacks, but also made my graduate school life much more fun and enjoyable with the endless kitchen runs and the actual runs. I would like to thank the members (and honorary members) of the CCCP group: Hyunchul Park, Mojtaba Mehrara, Amir Hormati, Shantanu

Gupta, Shuguang Feng, Ganesh Dasika, Amin Ansari, Sangwon Seo, Mark Woh, Yongjun Park, Jeff Hao, Po-Chun Hsu, Mehrzad Samadi, Ankit Sethia, Gaurav Chadha, Anoushe Jamshidi, Daya Khudia, Andrew Lukefahr, Janghaeng Lee, Jason Jong Kyu Park, Shruti Padmanabha, Silky Arora, and John Kloosterman.

Above all, the utmost of thanks go to my dear family. Whatever I am today, I exist thanks to the unconditional love and support of my parents, Hang Cheol Cho and Young Soo Kim. Learning how things work from my late grandfather, Myoung Kwon Kim, made me not hesitate a moment to choose becoming an engineer. I still remember the day my uncle, Joo Youn Kim, taught me how to use Norton Commander on MS-DOS 5.0 and such days led me to take computer engineering as my major. Finally and most importantly, I truly thank my wife, Hyo Jin Lee, for turning my life into the happiest one.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES x

ABSTRACT xi

CHAPTERS

1 Introduction 1
- 1.1 Correctness Challenges 4
- 1.2 Efficiency Challenges 5
- 1.3 Contributions 7

2 Practical Lock/Unlock Pairing 9
- 2.1 Introduction 9
- 2.2 Background and Motivation 12
  - 2.2.1 Gadara 12
  - 2.2.2 Challenges for Lock/Unlock Pairing 13
- 2.3 Static Lock/Unlock Pairing Analysis 17
  - 2.3.1 Simple Example of Analysis Flow 17
  - 2.3.2 Mapping Lock to Set of Corresponding Unlocks 20
  - 2.3.3 Path Condition Calculation 21
  - 2.3.4 Checking Lock/Unlock Pairing 22
  - 2.3.5 CFG Pruning 23
- 2.4 Inter-procedural Analysis 24
  - 2.4.1 Proximity-based Callgraph Partitioning 26
  - 2.4.2 Extending Lock/Unlock Pairing for Inter-procedural Analysis 26
- 2.5 Dynamic Checking 28
  - 2.5.1 Checking Lock-to-Unlocks Mapping 30

2.5.2 Checking Semiflow Property . . . 31
2.6 Experimental Results . . . 32
2.6.1 Effectiveness of Static Analysis . . . 32
2.6.2 Runtime Overhead of Dynamic Checking . . . 35
2.6.3 Assumption Violation . . . 37
2.7 Related Work . . . 38
2.8 Summary . . . 43

3 Dynamic Core Boosting and Per-Core Power Gating . . . 44
3.1 Introduction . . . 44
3.2 Motivation and Background . . . 48
3.2.1 Low Utilization of Asymmetric CMPs . . . 48
3.2.2 Core Boosting . . . 51
3.2.3 Per-Core Power Gating . . . 52
3.3 Core Boosting Assignment . . . 53
3.3.1 Modeling and Problem Formulation . . . 55
3.3.2 Assignment for Data Parallel Programs . . . 57
3.3.3 Assignment for Pipeline Parallel Programs . . . 59
3.4 Synchronization-Aware Dynamic Core Boosting . . . 61
3.4.1 System Overview . . . 61
3.4.2 DCB Architecture . . . 62
3.4.3 DCB Compiler . . . 64
3.4.4 DCB Runtime Subsystem . . . 67
3.5 Synchronization-Aware Per-Core Power Gating . . . 68
3.5.1 Operating System Support . . . 68
3.5.2 Profiling-based Selective Power Gating . . . 69
3.5.3 Wakeup Hint and Prefetching . . . 70
3.6 Evaluation Methodology . . . 70
3.6.1 DBT-based Performance Evaluation . . . 71
3.6.2 Evaluation of Energy Saving . . . 74
3.7 Experimental Results . . . 74
3.7.1 Accuracy of DBT-based Performance Evaluation . . . 75
3.7.2 DCB Performance Improvement . . . 76
3.7.3 Energy Saving of Synchronization-Aware Power Gating . . . 80
3.8 Related Work . . . 82
3.8.1 Performance Asymmetry in CMPs . . . 83
3.8.2 Dynamic Adaptation of Core Performance . . . 83
3.8.3 Thread Criticality Assessment . . . 84
3.9 Summary . . . 86

4 Instrumentation Sampling for Lightweight Profiling . . . 88
4.1 Introduction . . . 88
4.2 Background . . . 92
4.2.1 Dynamic Binary Instrumentation . . . 92
4.2.2 Overview of DynamoRIO . . . 93

4.3 Instrumentation Sampling . . . 94
4.3.1 Context Switch . . . 96
4.3.2 Temporal Unlinking and Relinking of Fragments . . . 97
4.3.3 Multi-threaded Programs . . . 97
4.3.4 Summarizing Profile Data . . . 99
4.4 Pre-populating Software Code Cache . . . 99
4.4.1 Finding Basic Block Headers . . . 100
4.4.2 Affinity-based Pre-population . . . 101
4.5 Performance Evaluation . . . 102
4.5.1 Experimental Configuration . . . 104
4.5.2 Edge Profiling . . . 105
4.5.3 Performance Overhead . . . 106
4.5.4 Profiling Accuracy . . . 108
4.6 Related Work . . . 110
4.7 Summary . . . 113

5 Conclusion . . . 115
5.1 Summary . . . 115
5.2 Future Work . . . 117

BIBLIOGRAPHY . . . 119

LIST OF FIGURES

Figure

1.1 Speedup of PARSEC [11] benchmarks with varying number of threads compared to single thread executions. 6
1.2 CPU cycles spent blocked for synchronization operations. 7
2.1 Infeasible path example 10
2.2 Challenges for lock/unlock pairing 14
2.3 Simple example of lock/unlock pairing. 18
2.4 Finding unlock set corresponding to lock 19
2.5 Calculating path conditions 22
2.6 Example of CFG pruning. 25
2.7 Example of uncaught infeasible path. 28
2.8 Instrumentation wrapper for lock and unlock 30
2.9 Example of unpaired lock due to type mismatch. 33
2.10 Runtime overheads of dynamic checking. 36
2.11 Incorrectly paired lock due to pointer problem. 39
3.1 Slowdown caused by performance asymmetry. 48
3.2 CPU time wasted for synchronization. 50
3.3 Modeling of workload imbalance and core boosting. 55
3.4 Dynamic Core Boosting system overview. 61
3.5 Example of progress reporting instrumentation. 65
3.6 Core boosting emulation with dynamic binary translation. 72
3.7 Errors in the simulated execution time of the performance asymmetry evaluation platform. 75
3.8 Normalized execution time of Heterogeneous, Reactive, and DCB. 76
3.9 Synchronization overheads of Heterogeneous, Reactive and DCB. 78
3.10 Arrival time of each thread for blackscholes. 79
3.11 Overhead of synchronization-aware per-core power gating. 80
3.12 Impact of optimization for streamcluster. 81
3.13 Energy savings of synchronization-aware per-core power gating. 82
4.1 Control transfer for instrumentation sampling. 95

4.2 Traditional vs. DynamoRIO's basic blocks. 101
4.3 Overhead of edge profiling. 103
4.4 Execution time overhead of the instant profiling framework across five configurations of *(duration / frequency)*. 104
4.5 Computational overhead of the instant profiling framework across five configurations of *(duration / frequency)*. 104
4.6 Effect of pre-populating a software code cache. 106
4.7 Edge profiling accuracy of the instant profiling framework across five configurations of *(duration / frequency)*. 107
4.8 Asymptotic edge profiling accuracy. 110

LIST OF TABLES

Table

2.1 Number of annotations needed to model 12
2.2 Coverage of static lock/unlock pairing analysis 28
2.3 Number of Basic Blocks before/after Pruning 35
2.4 Number of Unlocks in Corresponding Unlock Sets 37
3.1 Simulated architecture details. 74

ABSTRACT

INTELLIGENT MANAGEMENT OF INTER-THREAD

SYNCHRONIZATION DEPENDENCIES FOR CONCURRENT PROGRAMS

by

Hyoun Kyu Cho

Chair: Scott Mahlke

Power dissipation limits and design complexity have made the microprocessor industry less successful in improving the performance of monolithic processors, even though semiconductor technology continues to scale. Consequently, chip multiprocessors (CMPs) have become a standard for all ranges of computing from cellular phones to high-performance servers. As sufficient thread level parallelism (TLP) is necessary to exploit the computational power provided by CMPs, most performance-aware programmers need to parallelize their programs.

For shared memory multi-threaded programs, synchronization mechanisms such as mutexes, barriers, and condition variables, are used to enforce the threads to interact with each other in the way the programmers intended. However, employing synchronization oper-

ations in both correct and efficient way at the same time is extremely difficult, and there have been trade-offs between programmability and efficiency of using synchronizations.

This thesis proposes a collection of works that increase the programmability and efficiency of concurrent programs by intelligently managing the synchronization operations. First, we focus on mutex locks and unlocks. Many concurrency bug detection tools and automated bug fixers rely on the precise identification of critical sections guarded by lock/unlock operations. We suggest a practical lock/unlock pairing mechanism that combines static analysis with dynamic instrumentation to identify critical sections in POSIX multi-threaded C/C++ programs. Second, we present Dynamic Core Boosting (DCB) to accelerate critical paths in multi-thread programs. Inter-thread dependencies through synchronizations form critical paths. These critical paths are major performance bottlenecks for concurrent programs, and they are exacerbated by workload imbalances in performance asymmetric CMPs. DCB coordinates its compiler, runtime subsystem, and architecture to mitigate such performance bottlenecks. In addition, we propose exploiting synchronization operations for better energy efficiency through dynamic power management, while maintaining performance. Finally, combining instrumentation sampling with dynamic binary translation is suggested for low overhead profiling.

Although the works presented in this thesis address a variety of issues related to synchronizations from correctness to performance and energy efficiency, they are all inspired by the same observation that neither a static nor a dynamic approach is sufficient for intelligent management of synchronizations. Based on this observation, we combine compiler techniques to analyze programs without burdening the execution of programs and runtime techniques to adjust the execution with more accurate information. By applying this theme

of hybrid static/dynamic mechanism to managing synchronization dependencies, we explore the possibility of increasing programmability and efficiency of concurrent programs.

CHAPTER 1

Introduction

The semiconductor industry continues to scale down the size of individual transistor according to Moore's law. Increasing clock frequencies and exploiting instruction level parallelism (ILP) have been the main methodologies to turn the technology scaling into application performance gains for more than four decades. In recent years, however, CPU clock frequencies have flattened out due to power dissipation and thermal constraints. Similarly, design complexity and verification issues inhibit computer architects from exploiting more ILP. For these reasons, improving the performance of monolithic processors has become less successful, and chip multiprocessors (CMPs) have grown into a mainstream paradigm to improve application performance with the increased transistor counts, for all ranges of computing from cellular phones to high-performance servers. Since CMPs require sufficient thread level parallelism (TLP) to benefit from the provided computing power, most performance-conscious programmers face increasing pressure to parallelize their programs.

In shared-memory multi-threaded programs, which is the most prevalent programming model for CMPs, threads communicate with each other by reading from and writing to the

shared memory. The value read from the shared memory changes depending on the order of memory accesses to the location, and the order of memory accesses is decided according to the interleaving of threads. Therefore, controlling the thread interleaving is very important for making multi-threaded programs work in correspondence with the original intention, and programmers limit the legal interleavings of threads using synchronization operations such as mutexes, barriers, and condition variables.

While synchronization operations play an important role for multi-threaded programming by enforcing the threads to interact with each other in the way that the programmers intended, employing them in both correct and efficient way at the same time is very difficult. Faulty employment of synchronizations can result in errors (atomicity violations and order violations) or nontermination (deadlock). Such defects are collectively called concurrency bugs, and they are significantly more difficult to detect and fix than other types of bugs because they only manifest depending on specific thread interleavings.

Naïve placement of synchronization operations can also cause overly serialized executions. Synchronization operations sometimes can block the execution of threads and make them to wait until some conditions are satisfied. This blocking is necessary to prohibit illegal thread interleavings. However, blocking also limits the concurrency of parallel programs. If blocking is enforced more than necessary, it can be a performance bottleneck. Eyerman et al. [35] investigates a number of parallel benchmarks and discovers that synchronizations is the primary roadblock that prohibits the most number of benchmarks from scaling to many threads.

Programmers often make trade-offs between the programmability and the efficiency for employing synchronization operations. Since the fundamental purpose of synchronizations

is to limit a set of undesired thread interleavings, the more interleavings are prohibited, the more likely they cover all intended illegal thread interleavings. At the same time, however, larger set of illegal interleavings means higher possibility of blocking, limiting the concurrency of the execution. This trade-off can be seen in the familiar example of coarse-grained locking and fine-grained locking. Coarse-grained locking uses less number of mutexes and guards larger region of code with locks and unlocks. It is easier to use because it guarantees more atomicity and it is less prone to deadlocks. On the other hand, fine-grained locking is usually better for performance scaling because it serializes the execution less.

This thesis studies a collection of mechanisms that increase the programmability and efficiency of concurrent programs, without the intervention of programmers, by intelligently managing the synchronization operations. First, we focus on mutex locks and unlocks. Many concurrency bug detection tools and automated bug fixers rely on the precise identification of critical sections guarded by lock/unlock operations. We suggest a practical lock/unlock pairing mechanism that combines static analysis with dynamic instrumentation to identify critical sections in POSIX multi-threaded C/C++ programs. Second, we present Dynamic Core Boosting (DCB) to accelerate critical paths in multi-thread programs. Inter-thread dependencies through synchronizations form critical paths. These critical paths are major performance bottlenecks for concurrent programs, and they are exacerbated by workload imbalances in performance asymmetric CMPs. DCB coordinates its compiler, runtime subsystem, and architecture to mitigate such performance bottlenecks. Finally, we propose exploiting synchronization operations for better energy efficiency through dynamic power management, while maintaining performance.

In the following sections, the synchronization-related challenges for the programmabil-

ity and efficiency of concurrent programs are discussed. Next, we present our contribution for intelligent management of inter-thread synchronization dependencies. Finally, we describe the organization of this thesis.

1.1 Correctness Challenges

The foremost source that makes it so difficult to write correct concurrent programs is concurrency bugs. Concurrency bugs are synchronization defects allowing thread interleavings that are not expected by the programmers. They result in data corruption (atomicity violations and order violations) or nontermination (deadlock). They often survive thorough testing and introduce fatal errors in production, because they do not manifest very well depending on thread interleavings. In this section, we introduce the most common types of concurrency bugs [65].

Deadlocks: Synchronizations operations sometimes let threads wait for some conditions that must be satisfied by other threads, and these relations form inter-thread dependencies. If there exists a cycle of inter-thread dependencies, the set of involved threads cannot progress and the execution is said to be in a deadlock [88].

Atomicity Violations: Atomicity is a guarantee of isolation from different threads. If the data manipulation effect of a set of operations from one thread appears to be equivalent to that of a serial execution without intervention from other threads, the set of operations is told to bear atomicity. Programmers frequently expect atomicity of operations. For instance, consider the case where a counter variable needs to be incremented. It will consist of

three operations: load from the variable, increment, and store to the variable. These operations need to execute atomically, or the counter might have an unexpected value. Atomicity is usually enforced by mutex synchronizations. If the atomicity intention of a programmer is not satisfied by the actual implementation, it is called atomicity violation [66].

Order Violations: Programmers also expect certain orders among operations. For example, pointer dereferencing needs to occur after the allocation to the pointer but before freeing the pointer. Similarly, processing data is expected to happen after partitioning the input, followed by summarizing the results. Such orders are often placed with condition variables or barriers. If the programmer fails to enforce the order expectations, an order violation occurs [65].

1.2 Efficiency Challenges

Synchronizations are prone to become a performance bottleneck for multi-threaded programs since they limit concurrency. Naïve placement of them can overly serialize the executions. Eyerman et al. [35] studies the scaling bottlenecks of PARSEC [11] and SPLASH-2 [100] benchmarks, and identifies that synchronizations are the primary bottleneck for the largest number of benchmarks. In this section, we confirm the fact with some preliminary experiments on a machine with 32 cores.

Figure 1.1 shows the speedups of a subset of PARSEC [11] benchmarks with different number of threads normalized against their single thread execution. Ideally, an application that scales perfectly has to show the speedup equal to the number of threads. As can be

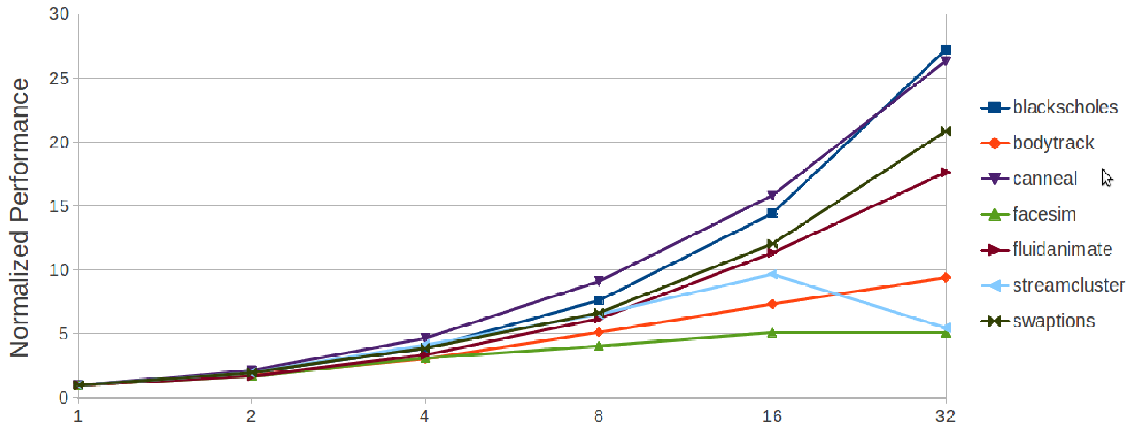

Figure 1.1: Speedup of PARSEC [11] benchmarks with varying number of threads compared to single thread executions.

seen, the benchmarks possess varying amount of scalability. While some programs such as *blackscholes* and *canneal* scale pretty close to the ideal, others like *streamcluster* or *facesim* do not scale very well. Regardless of whether they scale well or not, all of them show less performance improvement per thread as the number of threads increases.

We focus on how much processor time is wasted waiting for synchronization operations including mutex locks, condition variable waits, and barrier waits. We intercept every Pthread library calls by overloading `LD_PRELOAD` environment variable in Linux and measure waiting time for each operation. Figure 1.2 depicts the portion of time spent for synchronization. Comparing Figure 1.2 with Figure 1.1, we can see the benchmarks that spend more time for synchronization do not scale very well, which support the previous findings of Eyerman et al. [35]. Furthermore, this shows the potential of performance improvement if we can reduce the time spent for synchronization operations.

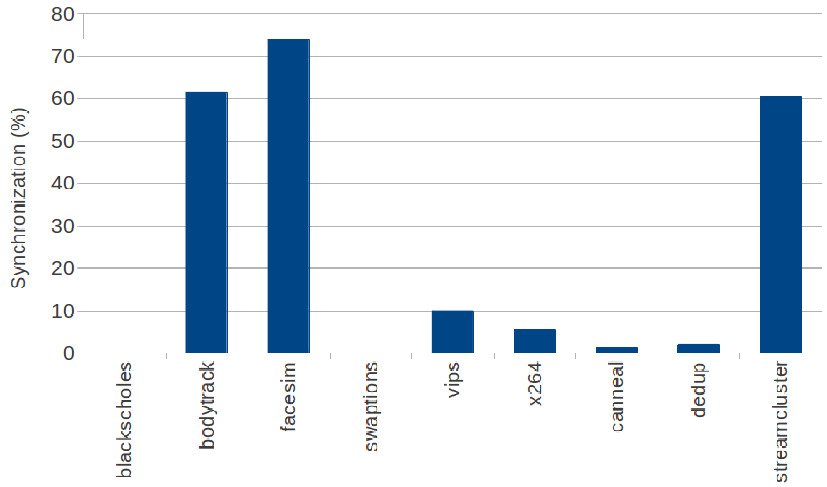

Figure 1.2: CPU cycles spent blocked for synchronization operations.

1.3 Contributions

Although the works presented address a variety of different issues related to synchronizations from correctness to performance and energy efficiency, they are motivated by one observation that neither a static nor a dynamic approach is sufficient for intelligently managing synchronizations. Based on such observation, we combine compiler techniques and runtime techniques to increase programmability and efficiency of concurrent programs. In particular, the main contributions offered in this dissertation are as follows:

- First, practical lock/unlock pairing mechanism for C/C++ is presented. This mechanism combines static analysis and dynamic instrumentation to identify critical sections in POSIX multi-threaded programs. It first applies a conservative inter-procedural path-sensitive dataflow analysis to pair up all lock and unlock calls. When the static analysis fails, our method makes likely assumptions about the pairing using common heuristics. These assumptions are checked at runtime using lightweight instrumenta-

tion [25].

- The second part of this dissertation proposes Dynamic Core Boosting (DCB), a software-hardware cooperative system that accelerate critical paths formed by inter-thread synchronization dependencies. DCB coordinates its compiler, runtime subsystem, and architecture for near-optimal assignment of core boosting. The compiler instruments the program with instructions to give progress hints and the runtime subsystem monitors their execution, enabling the DCB architecture to assign a core boosting budget for better performance. Improving energy efficiency through synchronization-aware per-core power gating is also presented [22, 23].

- The third part presents Instant Profiling, a lightweight flexible profiling mechanism used for the other parts of the dissertation. Instant profiling maintains low overhead with instrumentation sampling technique using dynamic binary translation. Instead of instrumenting the entire execution, instant profiling periodically interleaves native execution and instrumented execution according to configurable profiling duration and frequency parameters. It further reduces the latency degradation by pre-populating a software code cache [24].

The rest of this dissertation is organized as follows. Chapter 2 introduces the practical lock/unlock pairing mechanism. The Dynamic Core Boosting and synchronization-aware per-core power gating is described in Chapter 3. Instrumentation sampling for lightweight flexible profiling is presented in Chapter 4. Finally, Chapter 5 concludes the dissertation.

CHAPTER 2

Practical Lock/Unlock Pairing

2.1 Introduction

Most performance-aware programmers are experiencing ever growing pressure to parallelize their programs because uniprocessor performance has flattened out and multicore processors promise more cores in each successive hardware generation. Parallel programming, however, remains a daunting task for a variety of reasons. First of all, reasoning about concurrent events and synchronization is inherently challenging for human programmers, who think sequentially. In addition, concurrency bugs such as deadlocks, data races, and atomicity violations require global knowledge of the program. Finally, the nondeterministic execution of parallel programs makes these bugs hard to detect, reproduce, and fix.

There has been much effort to relieve the burden of parallel programming. For example, many automatic concurrency bug detection tools have been developed [84, 34, 66]. Automated bug fixing tools are also available [98, 53]. In addition, researchers are explor-

ing ways to adapt classical compiler optimization techniques for sequential programs to parallel programs [56, 87].

The aforementioned techniques often rely on or benefit from precise lock usage information, which is very difficult to obtain for languages without lexically scoped critical sections. For instance, static bug detection tools using *lockset analysis* [84] must map each statement to the set of locks held by the thread of execution. Infeasible paths such as the one illustrated in Figure 2.1 are a major source of false positives [34]. Automated bug fix tools often add locks to avoid deadlock [98] or restore atomicity [53], which may introduce new deadlocks if the usage of existing locks is unknown. Finally, it is well known that many compiler optimization techniques cannot be directly applied to concurrent programs [87]. Currently compilers only optimize code sections that do not involve any lock operation, which can be quite conservative [56]. Correct identification of critical sections allows better optimization for concurrent programs.

```
if ( x )
    lock(L);
...
if ( x )
    unlock(L);
```

Figure 2.1: Infeasible path example

Numerous static analysis approaches have been developed and many can be adapted to infer critical sections. In general, model checking based tools are precise but do not scale to practical programs [52], while scalable tools using specially designed algorithms are often imprecise [28, 19, 29]. For example, Saturn [29] is a scalable static analysis engine that is both sound and complete with respect to the user-provided analysis script. Writing a script that is sound and complete with respect to the target program, however, is as difficult as writing an analysis engine itself. The lock analysis script bundled with Saturn is neither

sound nor complete, most notably because it lacks global alias analysis.

In this chapter, we propose a practical lock/unlock pairing mechanism that combines dataflow analysis with dynamic instrumentation. Our interprocedural path-sensitive dataflow analysis is a variant of existing tools [28, 19]. It conservatively identifies lock acquisition and release pairs. When the analysis is uncertain, we use heuristics such as those based on structure types and location proximity to determine the pair. Finally, we instrument the target program with light-weight checking instructions to monitor whether the pairing is correct at run-time. When a violation occurs, feedback information is provided to revise the pairing.

This chapter makes several contributions. We present a static lock/unlock pairing analysis algorithm which yields accurate results in most cases. We develop a lightweight dynamic checking mechanism to ensure our analysis is correct. We demonstrate the effectiveness of our lock/unlock pairing mechanism including both static analysis and dynamic checking with real-world multithreaded programs such as OpenLDAP, Apache, and MySQL.

The remainder of the chapter is organized as follows. We first present the challenges with motivating examples in Section 2.2. Next, Section 2.3 describes how our static lock/unlock pairing analysis works, and Section 2.4 discusses how to extend the analysis for inter-procedural cases. We explain the dynamic checking mechanism in Section 2.5. Section 2.6 presents the experimental results and Section 2.7 outlines related work. Finally, we summarize the contributions and conclude in Section 2.8.

Benchmarks	Number of Annotations
OpenLDAP	90
MySQL	71
Apache	19

Table 2.1: Number of annotations needed to model

2.2 Background and Motivation

While our approach can help any tool that needs accurate static information about critical sections, we show its effectiveness in the context of deadlock avoidance. In this section, we briefly provide some background on dynamic deadlock avoidance in Gadara [98] and explain what kind of challenges exist for lock/unlock pairing.

2.2.1 Gadara

Gadara is a tool that enables multithreaded programs to avoid circular-mutex-wait deadlocks at runtime. The basic idea is to intelligently postpone lock acquisition attempts when necessary to ensure that deadlock cannot occur. It proceeds in the following phases. First, Gadara constructs a Petri net model from program source code using compiler techniques. Based on structural analysis of the model, Gadara synthesizes feedback control logic for each structural construct in the model that contributes to a potential deadlock. Finally, it instruments the control logic into the target program.

Given that the model is correct, Gadara automatically synthesizes maximally permissive controllers that delay lock acquisitions only if the program model indicates that deadlock might later result if the lock were granted immediately. Due to lack of accurate information about critical sections, however, Gadara's program analysis or control synthesis

could fail. In such cases, Gadara requires programmers to provide annotations. These annotations specify which unlocks match and that no mutex can be held at certain program points. Table 2.1 shows the number of annotations needed to model the benchmarks for Gadara. Providing annotations can be tedious, difficult, and error-prone even for programmers familiar with both the target program and Gadara.

2.2.2 Challenges for Lock/Unlock Pairing

This subsection discusses the major challenges that a static lock/unlock pairing analysis should address in order to model programs accurately in the context of deadlock avoidance. We illustrate the challenges via simplified code examples.

2.2.2.1 Infeasible Path

One of the challenges for static lock/unlock pairing is the ambiguity caused by infeasible paths. The traditional way in which compilers abstract programs' control path is to represent programs with control flow graphs (CFGs). Each vertex in a CFG represents a basic block and each edge corresponds to a branch from one basic block to another. Thus, a sequence of basic blocks connected with edges in a CFG represents a control path. Not all sequences, however, are actually possible control paths in program execution; some are *infeasible paths*.

Infeasible paths are a challenge for lock/unlock pairing because there can be cases where a lock is paired up with unlocks for all feasible paths but not paired up for some infeasible paths. This can be seen in the example of Figure 2.2 (a). If we assume the value of variable `flag` is not changed throughout the snippet, all the branches corresponding

```
1 :   if (flag)   lock(node->mutex);
2 :   while (condition) {
3 :       ...
4 :       if (flag)   unlock(node->mutex);
5 :       ...
6 :       node = node->next;
7 :       ...
8 :       if (flag)   lock(node->mutex);
9 :       ...
10:   }
11:   if (flag) unlock(node->mutex);
                    (a)

1 :   void callee(task *ptr) {
2 :       unlock(ptr->mutex);
3 :       ...
4 :       lock(ptr->mutex);
5 :   }
6 :
7 :   void caller(task *ptr) {
8 :       lock(ptr->mutex);
9 :       ...
10:       callee(ptr);
11:       ...
12:       unlock(ptr->mutex);
13:   }
                    (b)

1 :   lock(parent->mutex);
2 :   ...
3 :   if (condition1) {
4 :       lock(child->mutex);
5 :       ...
6 :       unlock(child->mutex);
7 :   }
8 :   ...
9 :   unlock(parent->mutex);
                    (c)
```

Figure 2.2: Challenges for lock/unlock pairing

to the `if` statements should follow the same direction. However, a naive static analysis would consider all possible combinations of branch directions, if it cannot correlate branch conditions. Another example of infeasible path is infinite loops, since we can assume a finite number of iterations for all reasonable executions. In the same example, if the analysis considers the infinite loop, the lock of line 8 might not be paired up.

This challenge is especially problematic for Gadara, whose models must have the semiflow property [99]. Intuitively, the semiflow property means that a mutex acquired by a thread should be released later in the model. It is not always satisfied, however, if we directly translate the CFG to a Gadara model due to the infeasible path problem as described above. The semiflow property is one of the most important characteristics of Gadara models, and it is the main reason why accurate lock/unlock pairing is important in the context of deadlock avoidance.

2.2.2.2 Spanning Function Boundaries

Another challenge arises from the fact that locks and unlocks need not reside in the same function. For widely used concurrent programs, it is not rare for locks and unlocks to span multiple levels of call chains. If a lock/unlock pairing analysis operates only within function boundaries it is not possible to pair up such cases.

Figure 2.2 (b) illustrates such a case. An intra-procedural lock/unlock pairing analysis would conclude that the lock in function `caller()` is paired up inside the function and the lock in function `callee()` is not paired up. However, actually the lock in line 8 is paired up with the unlock in line 2 and the lock in line 4 is paired up with the unlock in line 12, in this calling context.

It gets even more complicated since there can be many calling contexts. Gadara models function calls by substituting into the call site a copy of the callee's Petri net model [98], thus the model of one function can be analyzed differently depending on the calling contexts. Therefore, in order to handle this kind of cases, the lock/unlock pairing analysis should be inter-procedural and context-sensitive.

2.2.2.3 Pointers

Imperfect pointer analysis imposes another challenge. Mutex variables are usually passed to lock/unlock functions via pointers. Since only locks and unlocks on the same mutex variable pair up, it is important to figure out which lock pointers point to the same location and which pointers do not. As widely known, however, even state-of-the-art pointer analyses cannot provide perfect information. For some cases, they can only conservatively tell that two pointers *may* alias.

Figure 2.2 (c) illustrates how pointers can cause problems for lock/unlock pairing analysis. Assume that the pointer analysis concludes that the pointer `parent` and `child` may alias, which is the normal case for heap variables. In this case, although it is reasonable for a human programmer to pair up the lock in line 1 and the unlock in line 9, it is not trivial for a static analysis to reach the same conclusion.

In order to model concurrent programs accurately, the lock/unlock pairing analysis should work well under such circumstances with imperfect information about pointers. Furthermore, Gadara conservatively approximates mutex pointers based on types [98]. More specifically, it models the mutexes accessed by pointers, which are enclosed in the same type of structure, as one resource place. The lock/unlock pairing analysis can relieve the

impact of this kind of approximation.

2.3 Static Lock/Unlock Pairing Analysis

In Sections 2.3 and 2.4, we discuss how our static analysis pairs up locks and unlocks to cope with the challenges described in the previous section. Section 2.3 first covers the detailed steps for intra-procedural cases and we show how to extend it for inter-procedural cases in Section 2.4.

Our static lock/unlock pairing analysis is carried out in four steps. First, the analyzer extracts an enhanced control flow graph (CFG) from source code and prunes it. This CFG is augmented with information about function calls and branch conditions. It prunes the CFG for computational efficiency, leaving only relevant branches. Second, it maps each lock to a set of corresponding unlocks through dataflow analysis traversing the CFG in a depth first manner while managing lock stack data structures. Third, it calculates Boolean expressions that express the conditions under which each lock and unlock is executed. Finally, using a SAT solver [32], it examines whether all locks are paired up with unlocks on every feasible path. Section 2.3.1 shows how the analysis works with a simple example, then the rest of the section describes each of these steps in detail.

2.3.1 Simple Example of Analysis Flow

This section presents the conceptual flow of our lock/unlock pairing analysis with a simple example in Figure 2.3. In this example, the mutex acquired by the lock in line 3 is always released before the function `handle_task()` returns by either the unlock in line 7

```
1 :  int handle_task(task *job) {
2 :      if(job->has_mutex)
3 :          lock(job->mutex);
4 :      if(job->is_special) {
5 :          // Handle special case
6 :          if(job->has_mutex) {
7 :              unlock(job->mutex);
8 :              return result;
9 :          }
10:      }
11:      //Handle normal cases
12:      if(job->has_mutex)
13:          unlock(job->mutex);
14:      return result;
15:  }
```

Figure 2.3: Simple example of lock/unlock pairing.

or the unlock in line 13. However, directly translating the CFG of this example to Petri net violates the semiflow property due to infeasible paths as described in Section 2.2.2. Our analysis rules out the infeasible paths and gives accurate lock/unlock pairing results through the following process.

After pruning the CFG, the corresponding unlock set mapping decides that both the unlock in line 7 and the unlock in line 13 can release the mutex acquired by the lock in line 3. Then, the path condition calculation step determines the Boolean expressions that represent path conditions for each lock and unlock. In this example, path condition $(job \rightarrow has\_mutex \neq 0)$ must be true for the lock to be executed. Similarly, the unlock in line 7 has $(job \rightarrow is\_special \neq 0) \wedge (job \rightarrow has\_mutex \neq 0)$, and the unlock in line 13 has $(job \rightarrow is\_special = 0) \wedge (job \rightarrow has\_mutex \neq 0)$ as their path conditions. The analysis then translates them into Boolean expressions by assigning a Boolean variable to each branch condition. In order to encode the branch correlations into the expressions, this assignment process assigns the same Boolean variable to branch conditions that must have the same value. As a result, the Boolean expression of the lock is (x_1), and the unlock in

```
1 : // Apply analysis to all functions
2 : void traverse_function(fn)
3 :     // Traverse CFG to calculate GEN and KILL sets
4 :     traverse_bb(entry);
5 :     for(each lock discovered)
6 :         corresponding_unlocks[lock]
7 :                         = GEN[lock] - KILL[lock];
8 : end
9 :
10: // Compute GEN and KILL sets for all locks traversing
11: // CFG in a depth first manner while managing lock
12: // stack data structure
13: void traverse_bb(bb)
14:     for(each instruction s in bb in order)
15:         if(s is a lock)
16:             push s to lock stack;
17:         else if(s is an unlock)
18:             top = top element of lock stack;
19:             add s in GEN[top];
20:             for(each element e in lock stack, e!=top)
21:                 add s in KILL[e];
22:             pop from lock stack;
23:     for(each successor child of bb)
24:         traverse_bb(child);
25: end
```

Figure 2.4: Finding unlock set corresponding to lock

line 7 and the unlock in line 13 receive $(x_2 \wedge x_1)$ and $(\neg x_2 \wedge x_1)$, respectively.

The final step of the analysis is to check whether the lock and the corresponding unlocks pair up for all feasible paths. This can be done by determining whether the statement "if the path condition for the lock is true, then the disjunction of the path conditions for corresponding unlocks is true" is always true or not. In this example, the statement is interpreted as the Boolean expression $(\neg x_1) \vee (x_2 \wedge x_1) \vee (\neg x_2 \wedge x_1)$. In order to verify if it is always true, we apply a SAT solver on the negation of the Boolean expression. If it is unsatisfiable, then the statement is always true, and the lock and the corresponding unlocks are paired. Otherwise, they are not paired up. In this example, the negation of the Boolean expression is unsatisfiable and the lock is paired up with the unlock in line 7 and line 13.

2.3.2 Mapping Lock to Set of Corresponding Unlocks

Before applying the infeasible path analysis, this step groups corresponding locks and unlocks. More specifically, it maps a set of corresponding unlocks to each lock. We say an unlock corresponds to a lock if it can release the mutex acquired by the lock on any path. Since the mutex acquired by a lock can be released at different program points, we map a set of corresponding unlocks and not just a single unlock. This step is necessary because the same mutex can be acquired and released multiple times.

This analysis algorithm traverses the CFG in a depth first manner while managing a stack of locks for each mutex. The core analysis algorithm is given in Figure 2.4. Although we only show it for one mutex in this version, the actual analysis simultaneously works on all mutexes.

The underlying idea is to add an unlock to the corresponding unlock set of a lock, if there is a path in which the unlock follows the lock but there is no path in which there is another lock of the same mutex between the lock and unlock. The top element of the lock stack is the most recent lock that the traversal encountered, so it adds the unlock to the `GEN` set of the top element. Other elements in the lock stack are the locks that the traversal met before the last lock along the traversal, so it adds the unlock to the `KILL` sets of them. Ultimately, the corresponding unlocks of each lock are the unlocks that are in the `GEN` set but not in the `KILL` set.

2.3.3 Path Condition Calculation

This step calculates the Boolean expressions for path conditions that must be true for each lock and unlock to execute. It first calculates path conditions and translates them by assigning a Boolean variable for each branch condition. We define the path condition of a statement as the Boolean combination, i.e., AND($\wedge$), OR($\vee$), NOT($\neg$), of branch conditions, which must be true for the statement to be executed.

The core path condition calculation algorithm is illustrated in Figure 2.5. With this algorithm, the path condition of a statement is the path condition from the entry basic block to the basic block that the statement belongs to. The underlying idea of this algorithm is that the path condition from the CFG node `'src'` to `'dest'` is the disjunction (OR) of the conditions along the paths which go through `'child'`, for all children of `'src'`. This idea is reflected in line 18.

This algorithm uses caching and post dominator information for computational efficiency. Since there can be an exponential number of paths to the number of basic blocks, the naive recursive algorithm is not feasible for real programs. In order to avoid repetitive computation for the same path, it uses a path condition cache indexed by `(src,dest)` pair. In addition, it uses post dominator (PDOM) information as a shortcut, to simplify the resulting conditions.

After path condition calculation, the analysis translates the path conditions to Boolean expressions by assigning a Boolean variable to each branch condition. To reveal the branch correlations in the Boolean expressions, our analysis assigns the same Boolean variable to the branch conditions that must have the same value. This is possible by using global value

```
1 : // Recursively calculates path condition from
2 : // src to dest
3 : condition calculate_path_cond(CFG, src, dest)
4 :     // Consult path condition cache for efficiency
5 :     if (src, dest) is in cache
6 :         return condition from cache;
7 :     // Always reaches dest if it post-dominate src
8 :     if dest PDOM src
9 :         return TRUE;
10:     // Dead end
11:     if src has no successor
12:         return FALSE;
13:     // The control can reach dest following
14:     // each successor
15:     for(each successor c of src)
16:         cond1 = branch condition of branch (src->c);
17:         cond2 = calculate_path_cond(CFG, c, dest);
18:         condition = condition OR (cond1 AND cond2);
19:     put condition in cache with index (src, dest);
20:     return condition;
21: end
```

Figure 2.5: Calculating path conditions

numbering (GVN) and hashing them to map to Boolean variables.

2.3.4 Checking Lock/Unlock Pairing

Using the analysis results of the previous steps, this step finally verifies whether all locks are paired up with the corresponding unlocks on every feasible path. To achieve this goal, we use an open source SAT solver MiniSAT [32]. For each lock, through the previous steps, we have the set of corresponding unlocks and the relevant Boolean expressions for the path conditions of them and the lock. With these analysis results, to verify the statement "the lock is paired up with the corresponding unlocks on every feasible path" is equivalent to checking the proposition "if the Boolean expression for the lock is true, the disjunction of the corresponding unlocks' Boolean expressions is always true." Let L be the Boolean expression for the lock, and $U_1, U_2, ..., U_n$ be the Boolean expressions for the corresponding

unlocks. Then our analysis tries to check if the following expression is always true.

$$L \Rightarrow U_1 \vee U_2 \vee \ldots \vee U_n \tag{2.1}$$

or equivalently

$$\neg L \vee (U_1 \vee U_2 \vee \ldots \vee U_n) \tag{2.2}$$

Checking whether a Boolean expression is always true or not can be done with a SAT solver. If the expression is always true, its negation always evaluates false, which in turn implies that the negation of the expression is unsatisfiable. Therefore, we can check whether the lock is paired up with the corresponding unlocks by applying a SAT solver to the negation of (2.2), which is

$$L \wedge \neg U_1 \wedge \neg U_2 \wedge \ldots \wedge \neg U_n \tag{2.3}$$

2.3.5 CFG Pruning

One of the hurdles that the static lock/unlock pairing analysis must overcome is computational complexity. In real world server programs, the number of basic blocks in a function easily grows to several hundreds. In addition, the Boolean satisfiability problem is well known to be NP complete. For these reasons, we must carefully minimize the number of clauses in the Boolean expressions, in order to make our analysis scale to real programs. We achieve this by pruning the CFG.

Our analysis tool prunes the CFG without losing any relevant information needed for the analysis based on control dependence analysis [38]. Intuitively, CFG node X is control dependent on node Y if the outgoing edges from Y determine whether X is executed or

not, and control dependencies can be calculated by finding post-dominator frontiers in the CFG. Given this property of control dependence, when we calculate the path condition for a basic block X, we must consider the basic blocks on which X is control dependent, the basic blocks on which those basic blocks are dependent, and so forth. Therefore, if we calculate the control dependence closure for the basic block, the basic blocks in the closure are the only ones that are relevant for the path condition calculation. We calculate the control dependence closure by iteratively including basic blocks until it converges.

The CFG pruning algorithm works as follows. It starts with the basic blocks of interest as input. Then, it calculates the control dependence closure for them, i.e., the closure relevant basic blocks. Finally, we prune the CFG by maximally merging irrelevant basic blocks that are connected. The CFG pruning algorithm can be easily understood with the example in Figure 2.6. In this example, basic block 9 is the basic block of interest. It is control dependent on basic block 7; furthermore basic block 7 is control dependent on basic block 1. After calculating the control dependence closure $\{1, 7, 9\}$, the rest of the basic blocks can be merged if they are connected. The simplified CFG on the right results from pruning. By working on this pruned CFG, the path condition calculation and the resulting Boolean expressions get much simpler.

2.4 Inter-procedural Analysis

As discussed in Section 2.2.2.2, many lock/unlock pairs span function boundaries. In order to model concurrent programs for most cases, our lock/unlock pairing analysis must be inter-procedural and context-sensitive. In this section, we describe how to extend the

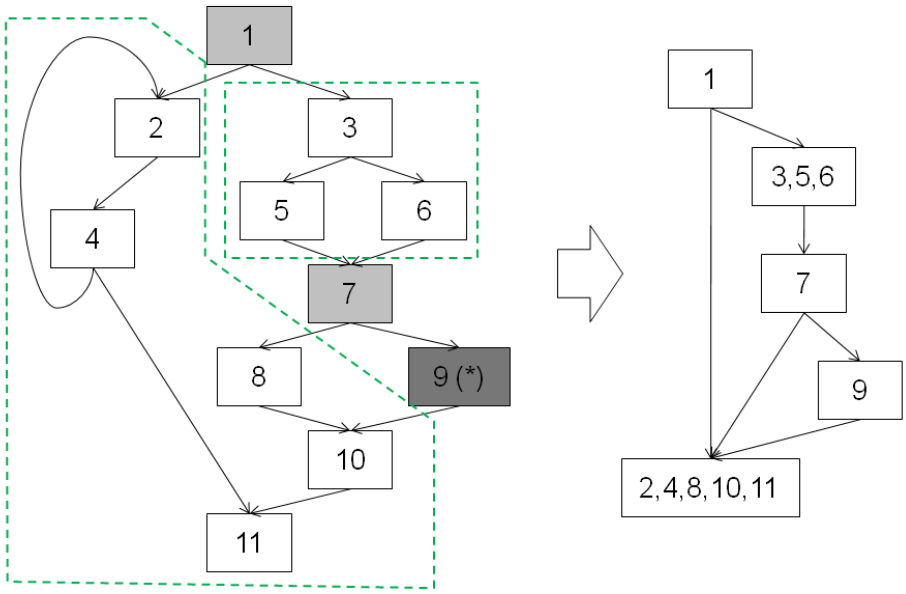

Figure 2.6: Example of CFG pruning.

analysis presented in the previous section for inter-procedural cases.

One straightforward way to make the analysis inter-procedural is a top-down approach that performs the analysis on the whole program CFG by conceptually replacing call instruction with the CFG of callee function at every call site, starting from the `main()` function. However, this can cause a computational complexity problem by producing an excessively large CFG to analyze.

Instead of flattening out the CFG for the entire program, we divide the problem into small pieces and perform the analysis on subgraphs in order to limit the analysis time. We first partition the callgraph with a proximity-based heuristic, and analyze the subgraphs in a bottom-up manner. We describe the details of this analysis in the following subsections.

2.4.1 Proximity-based Callgraph Partitioning

We made two observations while we were trying to manually pair up locks and unlocks across function boundaries. The first observation is that the calling contexts of a lock and paired unlocks differ from a lowest common ancestor in the callgraph with respect to the root node `main()`, in most cases. Suppose that a mutex acquired by a lock with the calling context of $main \Rightarrow f_1 \Rightarrow ... \Rightarrow f_n \Rightarrow f_{l1} \Rightarrow ... \Rightarrow f_{ln}$ is released by an unlock with the calling context of $main \Rightarrow f_1 \Rightarrow ... \Rightarrow f_n \Rightarrow f_{u1} \Rightarrow ... \Rightarrow f_{un}$ on a path, then the other unlocks, if any, that pair up with the lock usually have calling context that shares $main \Rightarrow f_1 \Rightarrow ... \Rightarrow f_n$ and f_n is a lowest common ancestor of them in the callgraph. The second observation is that the depths from locks and unlocks to the lowest common ancestor of the pairing context are relatively small (< 5) for most cases.

Based on the above observations we use a heuristic of proximity-based callgraph partitioning to keep the inter-procedural lock/unlock pairing analysis tractable. The partitioning algorithm works as follows. It starts from functions that have unpaired locks and follows upward the callgraph. It continues until it reaches a node that has the nodes with potentially pairing unlocks as descendants or a predefined depth threshold. Then, it cuts the subgraph from the node as a root. In this way, we can limit the size of Boolean expressions to be small enough to analyze.

2.4.2 Extending Lock/Unlock Pairing for Inter-procedural Analysis

For inter-procedural lock/unlock pairing, we apply the analysis described in Section 2.3 on the subgraph partitioned in the previous subsection. The inter-procedural lock/unlock

pairing analysis must handle function calls in a different way from the intra-procedural analysis, which just ignores function calls except locks and unlocks. The information about locks and unlocks in callee functions must be considered when the analysis meets a function call. Our analysis takes two different approaches to do so for mapping a lock to the set of corresponding unlocks and for path condition calculation.

Mapping a lock to the set of corresponding unlocks can be modified to be inter-procedural in a relatively straightforward way. It can be considered as conceptually inlining function calls. When it meets a function call it follows the CFG of the callee function. When the function returns it goes back to the caller function's CFG. Other than that, it is identical to the mapping algorithm explained in Section 2.3.2. It is a simple extension but it is enabled by the proximity-based callgraph partitioning.

On the other hand, path conditions are calculated in a bottom-up manner. In order to calculate the path condition that decides the execution of a lock, it first calculates the lock's path condition in the leaf node function that contains the lock. Then, following the context recognized in the partitioning, it calculates the path condition of the function call in its caller function, and its caller function, and so forth until it reaches the root function of the partition. These conditions get merged with a conjunction operator to finally calculate the context-sensitive path condition for the lock. After it calculates the context-sensitive path conditions for the locks and the unlocks, the remaining steps are identical to the intra-procedural analysis.

```
1 :  Connection *c = NULL;
2 :  for(; index < tblsize; index++) {
3 :    ...
4 :    if (connections[index].state == C_USED) {
5 :      c = &connections[index];
6 :      lock(&c->c_mutex);
7 :      break;
8 :    }
9 :  }
10:  ...
11:  if (c!=NULL) unlock(&c->c_mutex);
```

Figure 2.7: Example of uncaught infeasible path.

Benchmarks	LOC	Number of lock	Trivial	DFT	Our Approach				
					Statically Paired	Speculatively Paired	Total Paired	Unpaired	Static Analysis
OpenLDAP	271,546	357	110	267	319	34	353	4	152.7%
MySQL	926,111	499	147	428	463	26	489	10	211.8%
Apache	224,884	19	0	0	17	0	17	2	33.9%
pbzip2	4,011	3	0	1	2	1	3	0	23.4%
pfscan	752	11	8	10	10	1	11	0	50.0%
aget	835	2	2	2	2	0	2	0	43.8%

Table 2.2: Coverage of static lock/unlock pairing analysis

2.5 Dynamic Checking

Our static lock/unlock pairing analysis can be potentially incorrect in some cases due to the assumptions and heuristics it uses. In this section, we discuss these potential sources of incorrect analysis results and explain how our dynamic checking instrumentation can detect them.

One important reason why our analysis might yield potentially incorrect results is pointers as described in Section 2.2.2.3. Due to the limitations of the default memory dependency analysis, we augment it with generic aggressive refinements. Although the probability is very low, they can result in incorrect analysis results.

The second source of potentially incorrect analysis results is the proximity-based call-

graph partitioning heuristic. Although we could not find a real example, it is theoretically possible that one lock has two pairing unlocks whose lowest common ancestors with the lock differ. In that case, our partitioning algorithm can give an incorrect subgraph to analyze and end up with an incorrect analysis result.

Lastly, there are cases where our analysis maps unlocks correctly but cannot guarantee the lock is paired up with unlocks for all feasible paths due to the limitations of our analysis. An example of this case is shown in Figure 2.7. Our analysis can map the unlock in line 11 as the corresponding unlock of the lock in line 6. However, it cannot guarantee the lock is paired up for all feasible paths due to the lack of understanding about program semantics. A human programmer can easily figure out that the variable `c` is not `NULL` when the lock in line 6 is executed, thus the lock is paired up with the unlock in line 11 if the value of `c` is not modified in between. However, it is difficult for a static analysis to understand such program semantics. In this case, our analysis provides the mapping for the modeling as the best effort result. However, this type of best effort analysis result might be incorrect for other cases.

For these reasons, our lock/unlock pairing mechanism needs a way to verify whether all of the analysis results are correct or there exists any violation of the assumptions it made. In order to do that, we need to check two types of conditions. First, the mapping of unlocks to each lock should be checked. If the mutex acquired by a lock is released by an unlock that is not in the corresponding unlock set, it should be detected. Second, the semiflow requirement has to be checked. In other words, whether each lock is paired up with an unlock for all feasible paths or not is to be checked. In the following subsections, we discuss these two types of checking in detail.

```
1 :  lock_wrapper(mutex, callsite, callstack) {
2 :    lock(mutex);
3 :    LOCK_ID = get_id(callsite, callstack);
4 :    mutex_to_lock_id[mutex] = LOCK_ID;
5 :    ROOT_FID = SEMIFLOW_RESULT[LOCK_ID];
6 :    held_mutex[ROOT_FID].insert(mutex);
7 :  }
8 :  unlock_wrapper(mutex, callsite, callstack) {
9 :    UNLOCK_ID = get_id(callsite, callstack);
10:    LOCK_ID = mutex_to_lock_id[mutex];
11:    mutex_to_lock_id.erase(mutex);
12:    assert(LOCK_UNLOCK_PAIR[LOCK_ID][UNLOCK_ID]);
13:    ROOT_FID = SEMIFLOW_RESULT[LOCK_ID];
14:    held_mutex[ROOT_FID].erase(mutex);
15:    unlock(mutex);
16:  }
```

Figure 2.8: Instrumentation wrapper for lock and unlock

2.5.1 Checking Lock-to-Unlocks Mapping

We instrument all locks and unlocks to check whether the mapping of each lock to corresponding unlocks is correct or not. We first assign a unique ID to each lock and unlock. At runtime, the instrumented code manages a thread-local data structure that keeps the acquiring lock's ID of each mutex. Since the data structure is thread local, it does not need to synchronize with other threads to access the data structure. When an unlock releases the mutex, the instrumented code looks up the acquiring lock's ID of the mutex and checks whether its own ID is in the corresponding unlock set of the lock.

The IDs of locks and unlocks can be simply assigned as a unique number to each calling instruction for intra-procedural cases. However, if they are paired up by the inter-procedural analysis, we need to manage different IDs for different calling contexts even for the same lock or unlock. This is achieved by managing private call stacks. For the functions that appear in the subgraph analyzed by the inter-procedural analysis, we assign IDs and instrument the entrances and exits to push and pop the ID in the private call stack.

This call stack information is concatenated to the IDs of locks and unlocks in order to make it context sensitive.

Figure 2.8 is the pseudo code for our locks and unlock wrapper functions. It obtains context sensitive IDs of the locks used by the acquisition and release functions at lines 3 and 9, respectively. We verify whether the released lock is in the unlock set corresponding to the acquired lock at line 12.

2.5.2 Checking Semiflow Property

Another condition that we need to check dynamically is the semiflow property. As described in Section 2.2.2.1, the semiflow property guarantees that a mutex acquired by a thread will always be released later. With the static analysis we check this property by testing whether locks are paired with unlocks for all feasible paths. If the condition is not satisfied due to incorrect analysis, the dynamic checking should be able to detect it.

We also check this property by instrumenting locks, unlocks, and function exits. For this type of check, the instrumented code maintains the information about held mutexes indexed with the acquiring lock's ID. Again, these IDs are concatenated with call stack information for context-sensitive cases. We instrument the root node functions of the sub-graphs partitioned by the proximity-based partitioning to check whether it is holding any lock that should be paired up inside the calling context when it returns. This is done by checking whether the held_mutex[FID] set (kept in line 6 of Figure 2.8) is empty when the root node function (FID) returns.

2.6 Experimental Results

We have implemented the lock/unlock pairing mechanism including both the static analysis and checking instrumentation as a pass of the LLVM compiler infrastructure [59]. Our implementation operates on the LLVM intermediate representation and provides both analysis results and instrumented code. For the aggressive refinement of LLVM's memory dependency analysis, we use Gadara's type-based method for mutex pointers and memory profiling for other variables.

All of our experiments were executed on a 2.50GHz Intel Core 2 Quad machine with 8GB of memory running Linux 2.6.32. We evaluate the effectiveness of our lock/unlock pairing with Apache 2.2.11 web server [5], MySQL 5.0.91 database server [77], OpenLDAP 2.4.21 lightweight directory access protocol server [79], pbzip2 1.1.4, pfscan 1.0, and aget 0.4.

2.6.1 Effectiveness of Static Analysis

Table 2.2 shows the effectiveness of our static lock/unlock pairing analysis. The third column is the total number of locks and the fourth column is the number of locks trivially paired up in a basic block. We also compare our approach against depth first traversal (DFT) of control flow graph, which is used by previous static lockset-based tools such as RacerX [34]. Statically paired locks mean the number of locks that could be paired up with infeasible path analysis. Speculatively paired locks are the ones that our analysis could successfully map the corresponding unlock sets but could not guarantee pairing for all feasible paths due to the limitation described in Section 2.5. Thus the sums of the sixth

```
1 :  class THD {
2 :    struct st_my_thread_var *mysys_var;
3 :    ...
4 :    char* enter_cond(mutex_t* mutex) {
5 :      ...
6 :      mysys_var->current_mutex = mutex;
7 :      ...
8 :    }
9 :    void exit_cond(char* old_msg) {
10:      ...
11:      unlock(mysys_var->current_mutex);
12:      ...
13:    }
14:  };
15:  ...
16:  bool wait_for_relay_log_space(RELAY_LOG_INFO* rli) {
17:    THD *thd = rli->mi->io_thd;
18:    char *save_proc_info;
19:    ...
20:    lock(&rli->log_space_lock);
21:    save_proc_info = thd->enter_cond(&rli->log_space_lock);
22:    ...
23:    thd->exit_cond(save_proc_info);
24:    ...
25:  }
```

Figure 2.9: Example of unpaired lock due to type mismatch.

and seventh columns are the numbers of locks that our static analysis could pair up with unlocks. As can be seen in the table, our static analysis works effectively for nearly all of the cases. Overall, trivial pairing fails to handle 70% of locks and DFT fails to handle 20.5% of locks. By contrast, our approach handles all but 1.8% of locks—an eleven-fold improvement compared with DFT.

There are still unpaired locks, although the number of such cases is relatively small. There are three types of causes for these cases. First, there are inherently unpaired locks in the programs. Three unpaired locks of OpenLDAP are from one function, `ldap_new_connection()`, and in this category. When the function is called in certain contexts, these locks are paired up and our analysis can catch those cases. In other contexts, however, they are not paired

up and thus our analysis cannot pair them up.

The second category of unpaired locks is due to the type-based memory dependency analysis refinement that we use for mutex pointers. This refinement assumes that two mutex pointers do not alias if the types of wrapper structures enclosing the mutex variables are different. With this assumption our analysis cannot pair up a lock and unlocks if they have different types. The example in Figure 2.9 shows how this can cause a problem. In this example, the lock in line 20 and the unlock in line 11 are called on the same mutex, because the call of `enter_cond()` in line 21 saves the mutex in a pointer and passes it to `exit_cond()`. The problem is that the types of wrapping structure for the lock and the unlock are different. The wrapping type is `RELAY_LOG_INFO` for the lock and `st_my_thread_var` for the unlock. The type based memory dependency refinement would consider them not to alias, and consequently our lock/unlock pairing analysis cannot pair them. Among the unpaired locks of MySQL, eight of them are in this category.

The last cause of unpaired unlocks is function pointers. The current implementation of our lock/unlock pairing analysis cannot track the inter-procedural cases in which a function is called via a function pointer, since it uses the callgraph information which only puts edges for direct function calls. One of OpenLDAP's locks, two of MySQL's locks, and two of Apache's locks could not be paired up for this reason.

Static analysis time as a percentage of compilation time is presented in the last column of Table 2.2. Analyzing MySQL and OpenLDAP takes considerably longer than analyzing other benchmarks because they have more complex control flows and include more lock/unlock function calls. Table 2.3 presents the number of basic blocks in a function before and after CFG pruning, as described in Section 2.3.5. Our CFG pruning signifi-

Benchmarks	Before Pruning		After Pruning	
	Average	Maximum	Average	Maximum
OpenLDAP	20.19	818	2.22	80
MySQL	5.88	3513	1.18	112
Apache	12.12	465	1.02	13
pbzip2	6.16	431	1.06	10
pfscan	10.57	48	3.61	33
aget	12.11	35	1.83	16

Table 2.3: Number of Basic Blocks before/after Pruning

cantly reduces both average and maximum number of basic blocks, which is essential for the scalability of our static analysis procedure.

2.6.2 Runtime Overhead of Dynamic Checking

Figure 2.10 presents the runtime overheads of the dynamic checking instrumentation. For server programs, it is measured as the comparison of average response time to clients on the same machine. For `pbzip2`, `pfscan`, and `aget`, the execution times are compared. Four parallel clients and worker threads are used for the servers and the other programs, respectively. As can be seen in the graph, our checking instrumentation imposes very small overheads for the programs. The runtime overheads range from 0.5% to 3.4% and the average is 1.6%.

As discussed in Section 2.6.1, our static analysis yields three types of results: statically paired, speculatively paired, and unpaired. For both statically and speculatively paired locks, our framework instruments the checking mechanism presented in Section 2.5, whose major overhead comes from the executions of locks and unlocks. Therefore, even if our analysis does not work well so that it yields more speculatively paired locks, the runtime overhead would not be drastically increased. For unpaired locks, the current implemen-

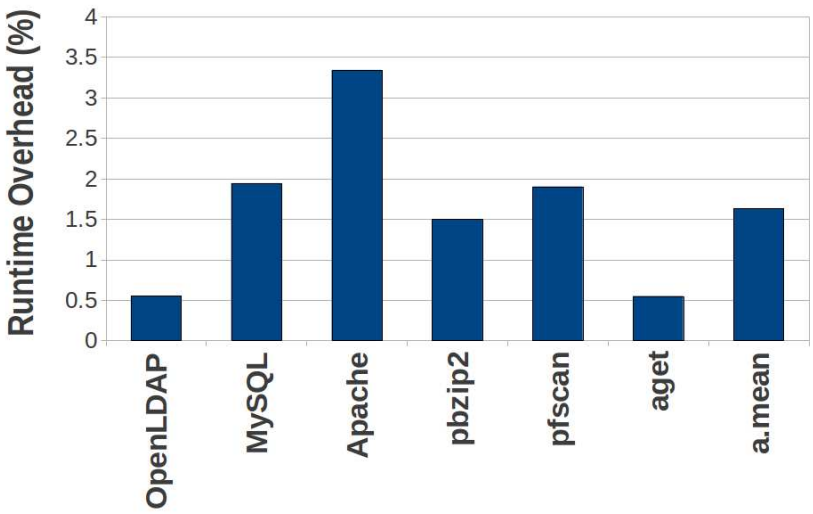

Figure 2.10: Runtime overheads of dynamic checking.

tation of our framework falls back to programmer annotations and does not add checking instrumentation. It is possible to add more heuristics to make guesses for unpaired locks, but the dynamic checking overhead would be still roughly proportional to the number of the executions of locks and unlocks even for those cases.

Compared to the native implementation of lock and unlock, our instrumentation slows down a pair of lock and unlock by roughly 18×. Thus, it is possible that our dynamic checking incurs excessive overhead if the target program locks and unlocks too many times without doing much work. However, it is not a common practice to make programs lock and unlock too often, and such programs would already suffer poor performance. Furthermore our current instrumentation implementation is a simple un-optimized use of the C++ STL library, and overheads can be further reduced by optimizing the implementation of instrumented code.

Benchmarks	Average	Maximum
OpenLDAP	1.37	7
MySQL	1.37	10
Apache	1.35	2
pbzip2	1.67	2
pfscan	1.09	2
aget	1.00	1

Table 2.4: Number of Unlocks in Corresponding Unlock Sets

2.6.3 Assumption Violation

Although the frequency is very low, our static lock/unlock pairing analysis can potentially yield incorrect results due to the assumptions and heuristics it uses as described in Section 2.5. As our analysis yields a set of unlocks for each lock, this imprecision means mistakenly including (false positives) or omitting (false negatives) an unlock. Both forms of imprecision can spell trouble, depending on how pairing information is employed. Our strategy is two-fold. The static analysis first strives to keep the unlock sets small, reducing false positives. Our dynamic checks then reliably find and report false negatives. It does not attempt to catch false positives, which would require exploring all possible execution paths.

Table 2.4 shows the size of corresponding unlock sets. For the most of cases, the unlock sets are quite small, minimizing the possibilities of false positives. However, there are a few cases which need upto ten unlocks to cover the many side exits of complicated control flows. We manually went over those cases, and did not find any false positives.

Once the instrumented dynamic checking detects a false negative, the information is fed back to the analyzer and the underlying client system revises the model. While we perform the experiments on the six programs, only one such case actually occurred for OpenLDAP

and the dynamic checking instrumentation detected it.

The code snippet that caused the violation is summarized in Figure 2.11. The cause of this incorrect analysis result is the type-based memory dependency analysis refinement that we use for mutex pointers. As opposed to the cases where different types for lock and unlock cause a problem, two distinct mutexes having same type is the problem in this case. The programmer's intention is that the lock in line 9 and the unlock in line 11 are called for different mutexes in the same iteration because `ei2` is supposed to point to one of the children of the node pointed by `eip`. Since both pointers have the same wrapper type, however, our mapping algorithm results in mapping the unlock in line 11 to the lock in line 9 and the unlock in line 16 to the lock in line 3. In real execution the mutex acquired by the lock in line 9 can be released by either the unlock in line 11 of the next iteration or the unlock in line 16 after breaking the loop. The lock in line 3 should also be paired up with both unlocks in line 11 and line 16. The instrumented checking code for the lock-to-unlocks mapping check detects this violation and reports the incorrect analysis result.

2.7 Related Work

In order to better model concurrent programs by pairing up locks and unlocks, we combine static analysis and dynamic checking. Since existing static analysis methods cannot provide a perfect solution to our purpose, we obtain best-effort analysis results with static analysis and check them at runtime to verify whether the results are correct. In this section, we first survey previous work on static analysis and dynamic monitoring techniques, focusing on the application of the lock/unlock pairing problem. Then, we provide possible

```
1 :  EntryInfo *eip, *ei2;
2 :  ...
3 :  for (lock(&eip->kids_mutex); eip; ) {
4 :    ...
5 :    // Search children in tree-like in data structure
6 :    ei2 = avl_find(eip->kids, ...);
7 :    ...
8 :    // Lock for next iteration
9 :    lock(&ei2->kids_mutex);
10:    // Unlock current node
11:    unlock(&eip->kids_mutex);
12:    eip = ei2;
13:    ...
14:  }
15:  ...
16:  unlock(&eip->kids_mutex);
```

Figure 2.11: Incorrectly paired lock due to pointer problem.

use case scenarios for our framework.

Static analysis. Existing static techniques applicable to the lock/unlock pairing problem can be largely divided into model checking methods that emphasize precision, and program analysis methods that emphasize scalability.

Software model checking has a long history. We recommend an excellent survey for the background on this subject [52]. Here we summarize several results relevant to this chapter. Classical model checking techniques model systems as *labeled transition systems* and verify properties specified in *temporal logic*. These techniques scale poorly for software verification due to the state explosion problem. Most software model checking tools are execution based and stateless. These tools systematically explore all program paths in hope to find bugs more quickly than stress testing [75, 14].

Abstract model checking scales to real software by mapping program states to an abstract domain [27]. As abstraction may not capture all the information needed to verify a property, when a counter-example is discovered, it is unclear whether it is genuine or

spurious due to abstraction. In this case, the abstraction can be refined to filter out spurious examples. Automated program abstraction and refinement are difficult, and the iterative process may not converge. In practice, automated abstract model checking methods are limited to small or special-purpose programs [42, 50].

In the area of static program analysis, many scalable dataflow analysis algorithms have been developed, which can be viewed as model checking with manually defined abstraction [85]. For example, Saturn [29] is a scalable analysis engine that is both sound and complete with respect to the user-provided abstraction, written in its Calypso language. This framework enables the programmer to manually refine and optimize the abstraction for each specific analysis task. Other scalable algorithms use carefully tuned heuristics that can be viewed as predefined abstraction. For example, ESP [28] is a path-sensitive analysis tool that scales to large programs by merging branches that lead to the same analysis state. The analysis is sound but incomplete with respect to this abstraction. Regarding the original program, however, manually defined abstractions are often unsound.

For example, the locking patterns in Figure 1(a) often confuses standard dataflow analysis algorithms integrated in tools designed for higher level applications [34, 98]. Both the locking analysis script bundled in Saturn and the ESP algorithm would identify the branch correlations easily if the code snippet is inside one function. But as both tools use function summaries for scalability, they can fail to infer correctly inter-procedural variations of the pattern if the function summary does not encode enough information. In this case, the analysis result can be sound but incomplete as missing information is often modeled by free variables with arbitrary values. On the other hand, the locking script in Saturn and ESP both ignore global alias, therefore the analysis result is unsound if the branching condition

`flag` is modified via a global pointer. Encoding sound and complete global alias information in function summaries is nontrivial [41]. Applications of Saturn often ignore alias analysis too [102].

As of today, we are not aware of any sound and complete program analysis tool that can verify the lock pairing property in large software such as Apache and OpenLDAP. Nevertheless, the analysis techniques in the previous program analysis tools have partially inspired the static analysis part of our work. For instance, we employ the infeasible path analysis similar to the ones used in [19, 28] and we also adopt caching analysis results for computational efficiency as RacerX [34] does.

Dynamic monitoring. Although not directly suitable for our problem, there has been a considerable body of work on monitoring the behavior of programs, especially in the context of profiling and bug detection. The main benefit of dynamic techniques is that they can closely collect information about program execution, which is difficult for static tools to infer.

Such tools as DynamoRIO [13], Pin [67], and Valgrind [78] provide generic instrumentation frameworks for dynamic monitoring. Through the comprehensive API of Pin and DynamoRIO, users can write their own monitoring client fitting their purpose, and Valgrind is widely used to detect memory bugs. In spite of the many optimizations they exploit such as code cache, branch linking, and trace building, however, they can impose a substantial amount of runtime overheads depending on what kind of code should be instrumented.

There are also dynamic monitoring techniques customized for specific purposes. LiteRace [69] and ReEnact [82] track concurrent programs' memory accesses to detect data races. AVIO [66] and AtomTracker [76] aim for atomicity violations. Our framework

shares the idea of reducing runtime overheads by customizing the type of tracking information with these tools. As opposed to these tools, however, our framework performs most of its analysis offline and uses dynamic checking only for confirmation.

Use cases. As mentioned in Section 2.1, our framework can benefit static bug detection tools and automated bug fix tools by providing more accurate information about critical sections. For instance, static bug detection tools using lockset analysis suffer false positives due to infeasible paths. RacerX [34] uses many heuristics and error ranking to mitigate the impact of such false positives. Our framework would help them prune invalid locksets and thus reduce the false positives. On the other hand, automated bug fix tools [53, 54] often add synchronizations to restore atomicity or order constraints, and they may introduce new deadlock if the usage of existing locks is unknown. AFix [53] sets timeout for the new synchronizations to avoid introducing deadlocks. Our framework can help them eliminate the timeouts and the potential chances of missing bugs.

Our framework can also promote compiler optimizations for concurrent programs. Currently compilers only optimize code sections that do not involve any lock operations, limiting the efficiency of the generated code. Joisha et al. [56] suggest extending the scope of optimizations beyond the synchronization-free regions by using procedural concurrency graph (PCG). With accurate lock/unlock pairing, they can further refine PCGs by reflecting the concurrency limited via mutexes. Consequently, this can provide more optimization opportunities.

2.8 Summary

We have proposed a practical lock/unlock pairing mechanism that combines an interprocedural analysis and dynamic checking for better modeling of critical sections in POSIX multithreaded C/C++ programs. We have demonstrated the effectiveness of our mechanism through experiments on six benchmarks including three large and complex server programs. Compared with depth-first traversal, our method reduces by 11$\times$ the number of statically unpaired locks. CFG pruning keeps problem size small so that compile time is low, and dynamic checking compensates for imperfections in our static analysis with modest overhead (at most 3.3%).

CHAPTER 3

Dynamic Core Boosting and Per-Core Power Gating

3.1 Introduction

Due to power dissipation limits and design complexity, the microprocessor industry has become less successful in improving the performance of monolithic processors, even with continued technology scaling. As a result, chip multiprocessors (CMPs) have grown into a standard for all ranges of computing from cellular phones to high-performance servers. Since CMPs require sufficient thread level parallelism (TLP) to benefit from the increased computing power, most performance-aware programmers face increasing pressure to parallelize their programs.

One lesson that programmers have learned from the long history of high performance computing is that increasing resource utilization results in better performance. As the multi-threaded programming model abstracts away the individual characteristics of each core, uniformly distributing workloads into threads has been considered an effective strategy to increase the utilization of CMPs.

Despite the best efforts of programmers to evenly divide workloads, it is very difficult, if not impossible, to perfectly balance workloads. Even for single program multiple data (SPMD) multi-threaded workloads with embarrassing parallelism, there exists implicit software heterogeneity among threads due to control flow divergence, non-deterministic memory latencies, and synchronization operations. Such software heterogeneity sometimes inhibits the parallel programs from effectively utilizing a larger number of cores.

The performance asymmetry of cores can notably exacerbate workload imbalance, and it is highly probable that we will have asymmetry in the future generations of CMPs for several reasons. First, heterogeneous multicore systems have been introduced by many researchers for better performance [7, 58] or saving power [57]. Heterogeneous multicores are also an effective way to trade die area to higher energy efficiency [68], and some commercial products [40] have already started implementing such designs.

Increasing core-to-core process variation also creates performance asymmetry in CMPs [92]. Process variation is the phenomenon where the process parameters of transistors, such as effective gate length and threshold voltage, diverge from their nominal value affecting the maximum operable frequency. The amount of within-die process variation is growing, as integrated-circuit technology keeps scaling down the size of individual transistors. With the rapidly developing emphasis on power and energy efficiency, lower supply voltages are preferred by chip designers and this makes the variation problem worse. Future microprocessors are likely to be heterogeneous across the working frequency of individual cores, since making all cores run at the frequency of the slowest core loses too much performance in the presence of large process variation.

One possibility for dealing with performance asymmetry in CMPs is to place the burden

of workload balancing on programmers or compilers. However, parallel programming itself is already difficult enough for programmers. Even if we assume that it was possible for compilers to exploit the heterogeneity for workload balancing, the portability issue would prohibit them from generating the customized code for more than one specialized setting of heterogeneity. Furthermore, often the performance asymmetry caused by process variation cannot be determined at compile time because it may vary from one chip to another even for the same model processor.

In this chapter, we propose software-hardware cooperative mechanisms to improve performance and/or energy efficiency for asymmetric CMPs. For better performance, Dynamic Core Boosting (DCB) tries to mitigate the workload imbalance problem. DCB relies on the hardware capability of accelerating individual cores through dynamic voltage and frequency scaling (DVFS) at a fine granularity to balance the workload across the asymmetric cores by boosting critical threads. With the limited resource to boost a subset of cores, DCB orchestrates its compiler, runtime subsystem, and processor cores for near-optimal assignment of the boosting budget. First, a target program is analyzed and instrumented by the compiler to include the instructions that provide progress hints. At runtime, the execution of the program is monitored by the DCB runtime subsystem. Finally, DCB selectively boosts the critical threads by using the information gathered by the instrumented code and the DCB runtime subsystem.

We also suggest adapting per-core power gating [60] for better energy efficiency. Per-core power gating is an effective way to save power by introducing a gate (or sleep transistor) between the power supply and each core. Compared to clock gating [61], power gating can save more energy by reducing leakage power to near zero but incurs a longer wake up

latency. Especially, losing private cache contents is the major obstacle that prevents per-core power gating from being deployed for finer granularity. In this chapter, we propose applying per-core power gating to idle cores due to workload imbalances in performance asymmetric CMPs. By combining early wake up and prefetching private cache contents, per-core power gating can save substantial amount of wasted energy with negligible extra overheads.

This chapter makes the following contributions:

- A theoretical background for the optimal assignment of core boosting.
- A cooperative system to balance workloads in asymmetric CMPs consisting of a compiler, runtime subsystem, and architecture.
- A novel mechanism to evaluate such systems with performance asymmetry and/or core boosting capability.
- A dynamic per-core power gating scheme to increase energy efficiency without performance loss.

The remainder of the chapter is organized as follows. Section 3.2 presents our motivation and provides the background of core boosting. Section 3.3 mathematically models the core boosting assignment problem and describes our algorithms at an abstract level. Section 3.4 explains the detailed implementation of the DCB system. Sections 3.6 and 3.7 present the methodology and the results of our evaluation. We discuss the related work in Section 3.8 and conclude the chapter in Section 3.9.

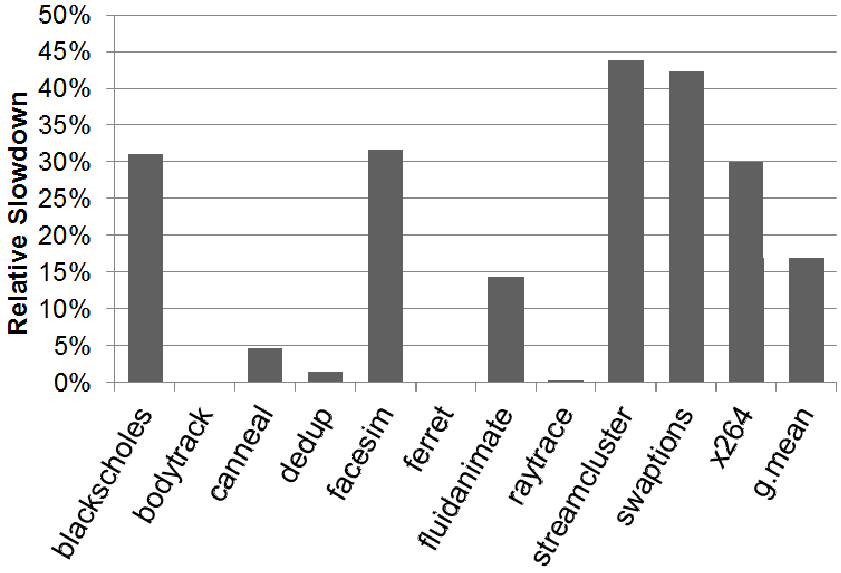

Figure 3.1: Slowdown caused by performance asymmetry.

3.2 Motivation and Background

While we can expect the performance asymmetry in CMPs to magnify the workload imbalance in multi-threaded programs, the exact effects on performance are not obvious. In this section, we present our motivation by showing the preliminary results on how much the asymmetry can affect the performance of multi-threaded benchmarks. Then, we provide the background of the hardware mechanism to accelerate and turn off individual cores.

3.2.1 Low Utilization of Asymmetric CMPs

We compare two simulated eight core systems to understand the performance impact of core asymmetry. The two systems work at the same average core frequency, but one has all eight cores operating at the same frequency and the other has varying frequencies. We assume a large variation in core frequencies ($\sigma/\mu = 30\%$, μ: mean, σ: standard deviation) as in Miller et al. [72], and the eight cores run at $(\mu - 1.5\sigma)$, $(\mu - 1.0\sigma)$, $(\mu - 0.5\sigma)$, μ, μ,

$(\mu + 0.5\sigma)$, $(\mu + 1.0\sigma)$, $(\mu + 1.5\sigma)$, respectively. The details of evaluation methodology are explained in Section 3.6.

Figure 3.1 presents the slowdowns of the asymmetric system compared to the symmetric one for the PARSEC 2.1 benchmark suite [11]. Most of the benchmarks are configured to have the same number of worker threads as the number of cores, except for those with pipeline parallelism. *dedup* and *ferret* are set to have one thread per pipeline stage. *x264* spawns the number of worker threads equal to the number of frames and there is no trivial way to change it with the harness of the PARSEC benchmark suite.

Even though the two systems have the same average core frequency, we can see that many of the benchmarks experience significant slowdown. Several benchmarks such as *streamcluster* and *swaptions* suffer the slowdown close to the worst core frequency. Some others, i.e., *bodytrack*, *ferret*, and *raytrace*, show almost identical performance to the homogeneous system on the other hand. The geometric mean of the slowdown for all benchmarks is 17%.

In order to understand what causes more slowdowns for some benchmarks than the others, we measure how much portion of CPU time in parallel sections is wasted on each type of synchronization. Figure 3.2 presents the measured portions. For each benchmark, the left bar shows the CPU time spent running on the homogeneous cores and the right bar represents the time on the asymmetric CMP. As seen in the graph, the benchmarks use different types of synchronizations as their main mechanism to control parallel execution, and the impact of performance asymmetry varies depending on the dominant synchronization pattern.

The simplest method is to spawn threads to work independently and join them at the

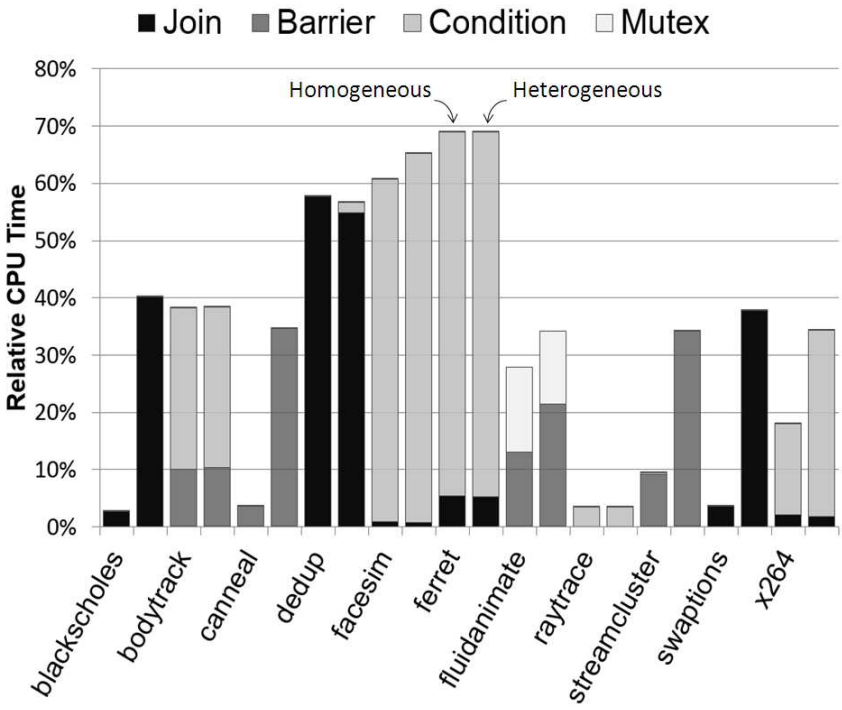

Figure 3.2: CPU time wasted for synchronization.

end. *blackscholes* and *swaptions* are in this category. Having similar structure, if the worker threads need to progress to the next stages together, they are synchronized with barriers. *canneal*, *fluidanimate*, and *streamcluster* use this type of synchronization patterns. For these two categories, the cores stay idle if their threads finish the tasks earlier than other cores, causing under-utilization of cores. Consequently, they are very likely to be affected by the asymmetry among cores.

Some benchmarks manage a pool of worker threads. When they need to execute in parallel, the main thread distributes tasks to the threads in the pool. After they finish the tasks, they stay idle waiting for the next task. The worker threads are usually synchronized with condition variables. If the workload distribution is determined dynamically, e.g., *bodytrack* and *raytrace*, they are less susceptible to workload imbalance due to asymmetric cores. On the other hand, *facesim* is

substantially affected by the asymmetry since the workload is equally divided once and assigned to the workers.

dedup and *ferret* adopt a pipeline parallel model. The worker threads run different stages of a pipeline and the data flows from one stage to another through a FIFO queue synchronized with condition variables. For this type of parallel program, the overall performance of the program is determined by the slowest stage. Accordingly, the performance is very sensitive to the stage-to-core scheduling for the asymmetric setting, but the average remains unchanged.

Finally, we see a great possibility of improving performance for asymmetric CMPs by balancing workloads. From the observations made above, many of the benchmarks are directly affected by the performance asymmetry. In addition, balancing the pipeline stages in the programs like *dedup* and *ferret* can yield performance benefits.

3.2.2 Core Boosting

Performance asymmetry among cores, combined with inter-thread dependencies formed by synchronization operations, causes a significant performance problem for multi-threaded programs as demonstrated above. We try to solve this problem by relying on the hardware capability of accelerating the subset of cores while staying in the power budget. Dynamic voltage and frequency scaling (DVFS) has been widely used for energy efficiency [2, 36]. Moreover, there have been several proposals that use dual power supplies for boosting individual cores [30, 72]. Dreslinski et al. [31] shows that very fast boosting transition ($<$ 10ns) can be achieved. Our system builds on such techniques for boosting cores at a fine

granularity.

While the idea of adopting fast core boosting for mitigating performance bottlenecks or reducing performance heterogeneity is not new [30, 72], the main contribution of our work lies in how to assign core boosting for higher performance with the same power budget. We first provide the theoretical background for the optimal assignment of core boosting. In order to achieve a close to the optimum solution, we propose a system that coordinates the compiler, runtime, and processor cores.

One important point to notice is that our assignment techniques are not limited to the specific core boosting technology. Although we assume a dual V_{dd}-based core boosting to demonstrate the effectiveness of our techniques in this chapter, our technique can be used in conjunction with any core acceleration mechanism with short enough transition time. Further differentiation from the previous proposals and more details of other feasible core boosting technologies are covered in Section 3.8.

3.2.3 Per-Core Power Gating

Power gating is a commonly used technique to turn off the power supply to a portion of circuit. Power gating can be applied at the varying granularities from functional units [101] and pipeline stages [45] to cores [60]. It is implemented by introducing a gate (or sleep transistor) between the power supply and the targeted circuitry. Compared to clock gating which only turns off the clock signal, power gating can save more power by reducing leakage current to near zero but incurs a longer wake up latency.

While per-core power gating has begun to make its way into commercial products [89],

it is only used for coarse time granularity (on the order of 100ms [60]) due to the long wake up latency. The reason why per-core power gating incurs this long latency is that the state of the core needs to be saved and restored. Although a recent work [51] achieves short latency (on the order of 10ns) per-core power gating and applies it to memory access stalls, they assume that the internal data is retained during power gating.

Our motivating observation is that synchronization stalls caused by workload imbalances in asymmetric CMPs are good targets for per-core power gating. In order to minimize the impact of losing private cache contents, we combine early wake up and prefetching.

3.3 Core Boosting Assignment

Given the core boosting capability and the limited boosting budget, how to assign the boosting budget is very important for overall performance. In this section, we show our core boosting assignment at an abstract level. At first, we describe the mathematical modeling of workload imbalance and core boosting. We then formulate core boosting assignment as an optimization problem and provide a theoretical solution. Finally, we explain our core boosting assignment algorithms for two commonly used parallelization practices: data parallel programs and pipeline parallel programs.

When programmers parallelize their compute intensive programs for better performance, they first have to decide how repeated computations can be divided into threads. If the computation is conducted on the multiple subsets of data and they can be potentially performed concurrently, data parallel structure is most commonly used. In this form of parallel programs, multiple worker threads are spawned to run same code on different, possibly over-

lapping, subsets of data. When some regions of code must be executed atomically, mutexes are used to guard the regions. In some cases, all worker threads should finish one phase of execution and be synchronized with each other before they start the next phase. Barrier waits are inserted between the phases for these cases.

For data parallel type of parallelism structure, software heterogeneity is implicit in the sense that worker threads run the same code. It does not always mean, however, that the amounts of computations are identical among the threads. Control flow divergence is the primary reason for such mismatch of computation. For example, if statements let different portions of code be executed depending on condition values. For some programs, even different number of loop iterations can be run depending on input data. Non-deterministic memory latencies are another important source of implicit software heterogeneity. Even though two threads are accessing the elements in the same array, one might hit and the other might miss in caches. Modern microprocessors usually have multiple levels of caches and accurately predicting the latency of each memory access is not possible. Lastly, synchronization operations also contribute to implicit software heterogeneity. For instance, when two threads are trying to acquire a mutex at the virtually same time, one might proceed immediately while the other waits until the mutex is released.

Another frequently used type of parallel structure is software pipelines. While the repeated computations can be executed concurrently in data parallel programs, some programs need to enforce orders among the computations performed on the different subsets of data. If different stages of computations can overlap preserving the orders, pipeline parallel structure is an option. For this type of parallel programs, multiple threads are spawned to execute the different stages of computations. Different stages are usually connected

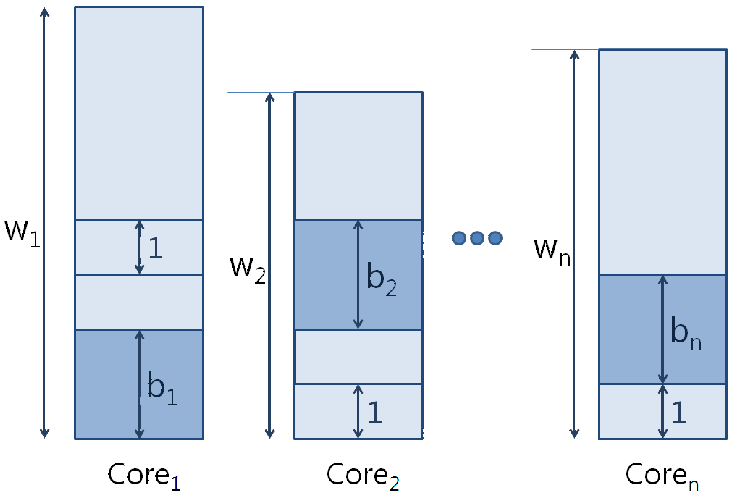

Figure 3.3: Modeling of workload imbalance and core boosting.

with FIFO queues and data elements flow from one stage to another through these queues. Condition variables are often used to synchronize the data flow.

Software heterogeneity is rather explicit in pipeline parallel programs, since different threads execute different codes. Since most modern microprocessors shows varying latencies depending on the types of instructions and the majority of them support out-of-order executions, statically balancing the execution time of different code is impossible even for homogeneous multicore processors. In addition, all sources of implicit software heterogeneity apply for pipeline parallel programs as well.

3.3.1 Modeling and Problem Formulation

Figure 3.3 depicts the modeling of workload imbalance and core boosting assignments with *n* cores. Without the loss of generality, this modeling assumes one workload for each core. If there are multiple threads running on a core, we can think of the total workloads of the threads as one workload. The assignment of core boosting can be changed after a

certain predetermined amount of time, called a quantum. Note that this boosting quantum is much shorter than the traditional OS scheduling quantum. This is possible as core boosting take place with very short transition time as mentioned in the previous section. Then, $w_1, w_2, ..., w_n$ denote the number of quanta taken to run each workload without any boosting on Core_1, Core_2, ..., Core_n. Each core can be accelerated to a different extent for the boosted mode, and $b_1, b_2, ..., b_n$ are the amount of acceleration. In addition, let $t_1, t_2, ..., t_n$ be the number of quanta where the boosting is assigned to each core.

Let us define the boosting budget, c, as the maximum number of cores that can be boosted at any quantum. For the best performance, c cores should be boosted every quantum, thus, it takes

$$T = \frac{1}{c} \times (t_1 + t_2 + ... + t_n) \tag{3.1}$$

boosting quanta to finish the execution. Moreover, $t_1, t_2, ..., t_n$ are bounded because a core can be boosted no more than once at any boosting quantum.

$$\forall 1 \leq k \leq n \ , \qquad 0 \leq t_k \leq T \tag{3.2}$$

The most important condition for this modeling to explain core boosting assignment is that every core must finish its workload within T quanta. For $\forall 1 \leq k \leq n$, Core_k runs t_k quanta boosted and $T - t_k$ quanta in normal mode, and it needs to finish its workload within T. Therefore, every t_k needs to satisfy the following inequality.

$$\forall 1 \leq k \leq n \ , \qquad (T - t_k) + b_k \times t_k \geq w_k \tag{3.3}$$

Since the number of boosted quanta for each core is an integer, core boosting assignment for the best performance is reduced to the integer linear programming [80] of minimizing T. Let us denote $P(w_1, w_2, ..., w_n)$ as the optimization problem of finding the minimal T and corresponding assignments $t_1, t_2, ...t_n$ when the workloads are $w_1, w_2, ..., w_3$.

3.3.2 Assignment for Data Parallel Programs

Although general integer linear programming is NP-hard, a solution can be quickly found with a greedy algorithm for our case. We will show that assigning the boosting budget to the cores with the largest remaining workload yields an optimal solution. We first prove the optimality of the greedy solution and then explain how we apply this to data parallel programs. For the simplicity of proof, c is assumed to be 1, but the same proof technique can be used for a larger boosting budget. The proof consists of two theorems.

Theorem 1 *If w_p satisfies $max(w_1, w_2, ..., w_n) = w_p$, then there exists an optimal solution for $P(w_1, w_2, ..., w_n)$ where $t_p \geq 1$.*

Proof. Suppose there exists an optimal solution, T' and $t'_1, t'_2, ..., t'_3$, where $t'_p = 0$. Since w_p is $max(w_1, w_2, ..., w_n)$ and $t'_p = 0$, the following can be derived from condition (3.3).

$$\forall 1 \leq k \leq n \ , \qquad T' \geq w_k \tag{3.4}$$

Then, let us find q such that $t'_q \geq 1$, and build another solution, T'' and $t''_1, t''_2, ..., t''_3$, by exchanging the values of t'_q and t'_p. Since we just exchanged two values, T'' remains the same as T'. From condition (3.4), this solution should also meets conditions (3.3). Therefore, T''

and $t''_1, t''_2, ..., t''_3$ is another optimal solution where $t'_p \geq 1$.

Theorem 2 *Let w_p satisfy $max(w_1, w_2, ..., w_n) = w_p$. If T' and $t'_1, t'_2, ..., t'_3$ with $t'_p \geq 1$ form an optimal solution for $P(w_1, w_2, ..., w_n)$, and T'' and $t''_1, t''_2, ..., t''_3$ form an optimal solution for $P(w_1 - 1, w_2 - 1, ..., w_{p-1} - 1, w_p - b_k, w_{p+1} - 1, ..., w_n - 1)$, then $T' = 1 + T''$.*

Proof. Since T' and $t'_1, t'_2, ..., t'_3$ satisfy condition (3.3), we can show they also satisfy the following condition with a little manipulation.

$$\{(T' - 1) - t'_k\} + b_k \times t_k \geq (w_k - 1), \quad if \;\; k \neq p \tag{3.5}$$

$$\{(T' - 1) - (t'_k - 1)\} + b_k \times (t_k - 1) \geq (w_k - b_k), \quad if \;\; k = p$$

Thus, $(T' - 1)$ and $t'_1, ..., t'_{p-1}, (t'_p - 1), t'_{p+1}, ..., t'_n$ also form a solution for $P(w_1 - 1, w_2 - 1, ..., w_{p-1} - 1, w_p - b_k, w_{p+1} - 1, ..., w_n - 1)$. With the similar manipulation, we can show that $(T'' + 1)$ and $t''_1, ..., t''_{p-1}, (t''_p + 1), t''_{p+1}, ..., t''_n$ form a solution for $P(w_1, w_2, ..., w_n)$ as well. Now, if we assume $T' > 1 + T''$, it contradicts that T' is an optimal solution since $(1 + T'')$ is a solution. Likewise, assuming $T' < 1 + T''$ contradicts that T'' is optimal because $(T' - 1)$ is a solution. Therefore, $T' = 1 + T''$.

The two proved theorems infer that boosting the core with the largest remaining workload at every quantum gives an optimal solution, hence the greedy algorithm will be optimal. Determining the remaining workload sizes at every quantum, however, is not possible in real systems. Consequently, we need a heuristic to decide which cores have the largest remaining workloads.

If we know the work progress ratio of each thread, we can approximately decide the thread with the least progress as the thread with the largest workload remaining. Although this heuristic is not always accurate, it works well when the threads are running similar amounts of workloads, which is usually the case for data parallel programs. As data parallel programs execute the same code for worker threads, we can instrument it to report work progress and assign a boosting budget to the cores with the least progress. The details of the program analysis and progress report instrumentation is explained in Section 3.4.3.

3.3.3 Assignment for Pipeline Parallel Programs

The heuristic used for data parallel programs does not work as well for pipeline parallel programs. It is primarily because pipeline parallel programs run different codes on different threads. It is difficult to measure progress consistently across threads running different codes. This makes it less likely that the thread with the least reported progress has the largest remaining work.

The synchronization pattern of pipeline parallel programs also makes it hard to apply the same technique. Multiple threads execute different stages of pipeline, and the data flows through the pipeline often using a FIFO queue. As it is difficult to perfectly balance workloads, some stages process data faster than the others. If one stage is significantly faster than its predecessor, the thread running the stage often waits on its input queue. Likewise, slow stages force their predecessors to wait. For this type of synchronization pattern, different stages make similar progress in terms of the number of data elements processed. Even though the same number of elements are remaining, however, faster stages

have less workload than slow stages. This invalidates the greedy solution and requires us to use a different approach for pipeline parallel programs.

We adopt an epoch-based approach with the observation that the relative speeds among threads alter much more slowly than the boosting quanta. When we look at the ratio of time spent working and blocked at a coarser grain than a boosting quantum (100 - 1,000x), the ratio of each thread tends to stay constant for the longer period of time. Our approach exploits the trend by assuming that the workload size of the previous epoch closely represents the current epoch. The details of how the workload sizes are approximated at every epoch is described in Section 3.4.4

At the end of every epoch, the core boosting assignment is calculated for the next epoch. Since the assignment takes place at runtime and heavy computation can nullify the performance gain, we need a simple solution. Instead of solving the integer linear programming in Section 3.3.1, the integerality condition is ignored assuming epoch size is large enough so that linear programming relaxation yields a close approximation. A heuristic based on Simplex algorithm [74] is used to quickly find an approximation with a minimal amount of computation. Assuming the minimum value of T exists on one of the extreme points, condition (3.2) and (3.3) states

$$\forall 1 \leq k \leq n \ , \quad t_k = 0 \quad or \quad t_k = \frac{w_k - T}{b_k - 1} \tag{3.6}$$

As a heuristic, w_k is then compared to $max(\frac{w_1}{b_1}, \frac{w_2}{b_2}, \ldots, \frac{w_n}{b_n})$ and assigned to 0 if it is smaller. Finally, the rest of t_ks can be directly calculated according to equation (3.6).

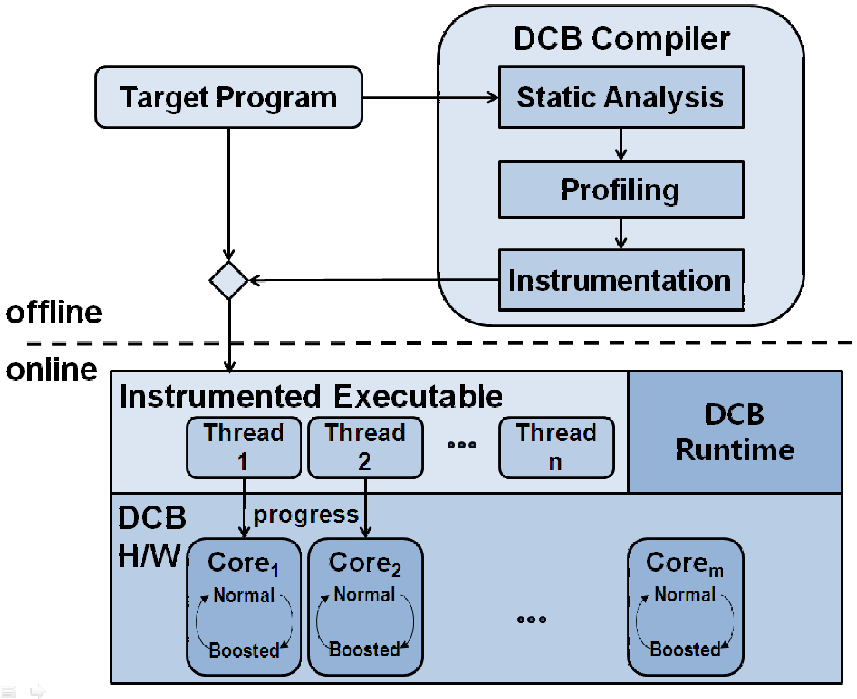

Figure 3.4: Dynamic Core Boosting system overview.

3.4 Synchronization-Aware Dynamic Core Boosting

This section describes how our Dynamic Core Boosting system (DCB) coordinates the compiler, the runtime subsystem, and the underlying core boosting architecture to obtain improved performance by balancing workloads.

3.4.1 System Overview

Figure 3.4 represents the overview of DCB. The DCB compiler takes a target program as an input. It first analyzes the parallelism structure and the control flow of the program, and generates profiling code. The profiling code then runs with a training input and produces profile data. Additionally, the DCB compiler makes decisions based on the static analysis results and the profile data to instrument the program with progress monitoring

code.

The generated executable runs on the DCB architecture along with the DCB runtime subsystem. In the DCB architecture, some cores can run in the boosted mode, which is faster than the normal mode. At every boosting quantum, the boosting manager in the DCB architecture decides which cores to run in the boosted mode while maintaining the boosting budget.

The instrumented code and the DCB runtime subsystem provide hints to the DCB architecture, with which the DCB architecture makes the boosting assignment decisions. For data parallel programs, the instrumented code reports the progress of each thread. At the end of every boosting quanta, the boosting manager chooses the threads with the smallest progress for boosting. DCB works differently for pipeline parallel programs. After every epoch, the DCB runtime subsystem calculates the desired boosting ratio among the threads to the DCB architecture, which stores the values for the next epoch. The boosting manager then probabilistically selects the cores to boost according to the boosting probability distribution.

3.4.2 DCB Architecture

While each core runs either in normal mode or boosted mode, it also takes hints and makes boosting assignments differently in two interface modes, namely progress mode and lottery mode, as briefly mentioned previously. The operating system takes this interface mode information with a flag for clone system calls when the threads are spawned. It stores the information and requests the DCB architecture to set the core in the proper mode

every time a context switch occurs. In addition, the thread ID and the thread group ID are utilized by the DCB architecture when a thread is scheduled in.

The progress mode is mainly for data parallel programs. Each thread reports its progress to the DCB architecture. After every boosting quantum, the boosting manager chooses *c* threads with the least progress in the same thread group to be boosted, where *c* is the boosting budget assigned to the thread group. The DCB architecture provides two non-privileged instructions so that the instrumented code can report its progress without the intervention of the operating system. PROGRESS_STEP_FORWARD increases the progress counter of the core by one, and SET_PROGRESS_TO(value) sets the progress counter to value.

The lottery mode works in a slightly different way. Each thread does not directly interact with the DCB architecture. Instead, the DCB runtime subsystem sets the desired boosting ratio among threads after every epoch. The boosting manager probabilistically choose *c* cores based on the ratio distribution in a similar manner to how the Lottery Scheduler [95] allocates resources. Pipeline parallel programs use the lottery mode to implement the assignment algorithm explained in Section 3.3.3.

All per thread information needed for the boosting assignment is stored in thread boosting table, which is managed by the operating system in the same way as page tables. The operating system and the DCB architecture can both access and modify the values in the thread boosting table. Moreover, the DCB architecture includes a cache for the thread boosting table as TLB for the page tables.

3.4.3 DCB Compiler

The main goal of the DCB compiler is to instrument the target program with the progress reporting instructions so that the boosting assignment algorithm described in Section 3.3.2 yields near optimal performance. In order to do so, the DCB compiler works in three steps: static analysis, profiling, and instrumentation.

At first, the DCB compiler statically analyzes the parallelism structure and the control flow of the target program. For the parallelism structure it investigates the starting and ending points of parallel execution in the main thread and the highest level functions executed in parallel. For the majority of programs, they are thread spawning function calls, thread joining function calls, and functions passed over to the thread spawning function calls, respectively. For some programs the DCB compiler cannot accurately gather the information. For example, the DCB compiler might be unable to disambiguate the function pointers passed over to the thread spawning calls. Moreover, non-standardized task starting and ending functions are used when the program manages a thread pool and send tasks to the pool for parallel execution. In those cases, the DCB compiler relies on the programmers' annotation specifying the information.

Once the parallelism structure is determined, the DCB compiler analyzes the control flow of the code regions that can run in parallel. At the highest level, these sections are the functions passed over to the thread spawning calls and the region of the main threads between the starting and ending points of parallel execution. There could be function calls in these regions, and the DCB compiler follows the call graph to analyzes the callees in turn. It stops following the call graph if there is a call through an ambiguous function

```
01   :  pthread_barrier_wait(barrier);
02(*):  SET_PROGRESS_TO(0);
03(*):  period = calc_period_LID_007(start, end);
04   :  for( i = start ; i <= end ; ++i ) {
05   :      ...
06   :      compute1(...);
07   :      if(side_exit) {
08(*):          SET_PROGRESS_TO(MAX_PROGRESS_007);
09   :          break;
10   :      }
11(*):      if(((end - i) % period) == 0)
12(*):          PROGRESS_STEP_FORWARD;
13   :  }
14   :  compute2(...);
15(*):  PROGRESS_STEP_FORWARD;
16(*):  period = calc_period_LID_008(max);
17   :  for( i = 0 ; i < max ; ++i ) {
18   :      compute3(...);
19(*):      if(((max - 1 - i) % period) == 0)
20(*):          PROGRESS_STEP_FORWARD;
21   :  }
22   :  pthread_barrier_wait(barrier);
```

Figure 3.5: Example of progress reporting instrumentation.

pointer or a cycle in the call graph. The barrier synchronization points are also included in the control flow information.

The DCB compiler generates the profiling code and runs it with a training input. It focuses on the loops in the parallel regions, using the control flow information gathered in the static analysis phase. The profiling code records the time spent in each loop and the iteration counts. Path profiling is also performed to discover the most frequent paths.

The last step exploits the profile data along with the static analysis results to instrument the code with the progress reporting instructions. In order to achieve the goal of the DCB compiler, all threads need to report progress at the points where they share the same progress ratio, regardless of what control path they take. One necessary condition is that all threads should go through the same number of progress reporting steps. It is straightforward for the counted loops with constant iterations. However, this is not always the case

and other types of loops make this condition difficult to meet. In other words, naively incrementing a progress counter after every iteration does not work because the total iteration counts might vary across the threads even for the same loops depending on the input.

For the counted loops with input dependent iteration counts, the DCB compiler inserts the code to calculate the number of iterations needed to be executed for the next progress reporting right before entering the loop. This number is then used as a progress reporting period inside the loop. The DCB compiler also instruments loop side exits to set the progress counter to the final progress value of the loop. The DCB compiler does not instrument uncounted loops. If an uncounted loop in a parallel region takes too much time, it might hurt the workload balancing capability of DCB. However, it is a very rare case and the programmers can insert the progress reporting code by themselves or turn the loop into a counted loop. For instance, consider an uncounted loop traversing a linked list. It is very difficult for a compiler to decide the number of iterations before entering the loop. However, the programmer can possibly transform it to a counted loop by adding an element count variable in the list header.

Another requirement for the instrumented code is that the frequency of progress reporting should be adequate. If the reporting granularity is too coarse, the boosting manager cannot get enough information to decide the most lagging thread. It should not be too fine because the progress reporting instructions can incur excessive overheads for this case. The DCB compiler tries to insert progress reporting instructions so that the execution times between them are roughly constant. It estimates the execution time with the instruction counts for straight-lined code regions. In the case of loops, it uses the profile data to calculate the approximate execution time per iteration.

Figure 3.5 shows a simple example of how the instrumented code would look like in source level. The lines marked with an asterisk presents the code inserted by the DCB compiler. calc_period_L007() in line 3 and calc_period_L008() in line 16 are the inline functions generated by the DCB compiler. They calculate the number of loop iterations needed to be executed for the next progress reporting. Constant values cannot be used in the same place because of programs that have different number of iterations across the threads, since the total progress counts should be equal for all threads. The generated inline functions calculate the progress reporting period so that all threads go through the same number of progress reporting steps. Another point to notice is the line 8. For the threads that exits the loop before it finishes the total iterations, the DCB compiler sets the progress counter to the maximum progress of the loop.

3.4.4 DCB Runtime Subsystem

The most important role of the DCB runtime subsystem is to provide the desired boosting ratio to the DCB architecture when the threads are running in lottery mode. The DCB runtime subsystem is idle for the most of the time and wakes up after every epoch. It then reads the per thread values of the CPU cycles. The DCB architecture has the dedicated hardware counters for per core CPU cycles and the operating system manages the per thread values in the thread boosting table. The DCB runtime subsystem estimates the workload size of each thread by comparing the current per thread CPU cycles with the last value. Then it calculates the desired boosting ratio of the threads according to the assignment algorithm described in Section 3.3.3.

Although the DCB runtime subsystem can be implemented as a shared library, it is preferable for it to be part of the operating system because it needs fast accesses to the thread boosting table. Since the thread boosting table is protected from unprivileged accesses, the DCB runtime subsystem should go through the system call interface if it is implemented as a shared library. This can cause a performance problem if the epoch size is too small.

3.5 Synchronization-Aware Per-Core Power Gating

We propose applying per-core power gating to idle cores due to workload imbalances in performance asymmetric CMPs. In order to minimize the impact of wake up latencies caused by per-core power gating, we only turn off idle cores when the thread is waiting for a long duration synchronization operation. Furthermore, we instrument the code to provide hint for waking up the thread on the power gated core before the signaling thread approach the synchronization point. Finally, the waken up thread executes prefetching code to warm up the private cache.

3.5.1 Operating System Support

In recent Linux kernels, futex system calls are used to impelement POSIX synchronization operations without busy waiting. When FUTEX_WAIT is called, the calling thread is placed in a kernelspace wait queue. The waiting threads are released by calling FUTEX_WAKE. If there is no active thread running on a core, the kernel issues halt instruction. The dynamic power management controller in the underlying hardware decides

which state the core should be placed in, depending on the core idle duration. The typical granularities of recent microprocessors are in the order of 100ms [60].

We extend futex system call to support selective power gating and givin wake up hints by adding two more operations: FUTEX_DEEPWAIT and FUTEX_HINT. FUTEX_DEEPWAIT is used to let the OS know this synchronization operation is a feasible target of per-core power gating. In addition, prefetching code is specified with a parameter. The OS manages one more waiting queue (deep sleep queue) for power gating targets, and turns off the core when all the threads are waiting in the deep sleep queue. We assume that the microprocessor exposes a direct control of the core power states to the kernel.

When FUTEX_HINT is called, a thread in the deep sleep queue is waken up. The thread then executes the prefetching code given by the FUTEX_DEEPWAIT call. If it finishes running the prefetching code before it is waken up by a FUTEX_WAKE call, then it waits on the normal wait queue. On the other hand, if it is waken up by a FUTEX_WAKE call before it finishes, the prefetching code is preempted and the thread starts running the code after the FUTEX_DEEPWAIT call.

3.5.2 Profiling-based Selective Power Gating

Due to the substantial amount of wake up latencies, only long enough synchronization stalls can save energy with per-core power gating. Our scheme selectively perform per-core power gating for those long enough synchronizations relying on profiling. Two versions of each synchronization operation are provided using FUTEX_WAIT and FUTEX_DEEPWAIT, repectively. The compiler selects which version to be used for each

call based on profiling data. For training runs, waiting time for each synchronization is measured. For synchronizations of which waiting times exceed a threshold with decent probability, the synchronization operation is replaced with the corresponding version with FUTEX_DEEPWAIT calls.

3.5.3 Wakeup Hint and Prefetching

For the synchronization call that is replaced with the deep sleep version, the corresponding operation to give wake up hint is instrumented. For example, pthread_barrier_hint() is inserted to give hint for the power gating version of pthread_barrier_wait(). pthread_barrier_hint() tracks the number of threads reached the hinting point, and broadcasts FUTEX_HINT when the last thread arrives. The placement of the hinting operation is also determined based on profiling.

The current implementation of our scheme relies on the programmer for prefetching code. The programmer provides prefetching function as a parameter for the deep sleep version of synchronization operations. When a thread is waken up from deep sleep by a hinting call, the specified prefetching code is executed.

3.6 Evaluation Methodology

This section describes the evaluation methodology that we use in order to measure the performance improvement of DCB and the energy saving of synchronization-aware per-core power gating.

3.6.1 DBT-based Performance Evaluation

As the system level interactions among threads are very important, the evaluation of DCB is different from the evaluation of other microarchitectural features. This difference makes the traditional evaluation approach of using cycle-accurate simulation an infeasible option for our purpose. DCB makes boosting assignment decisions based on the relative orders of thread progressions, and synchronization operations are critical to these orders. For instance, let us consider a situation where two threads are competing for a mutex. One of them is about to enter a long critical section and the critical section of the other is short. A slight difference of arrival time to the critical section can make a huge difference in the progress of the threads after they both exit from the critical sections. Moreover, even the execution path might change depending on the order of events [39]. Sampling [81] based simulation would not yield meaningful results as the interactions among threads are not considered. Trace-driven simulation that separates functional and timing simulation might not be accurate either.

Without sampling or trace-driven mechanisms, cycle-accurate simulators are too slow to evaluate the performance of DCB. The entire execution of the programs from the beginning to the very end must be measured since the interactions among threads are critical. This makes it very difficult, if not impossible, to test DCB on cycle-accurate simulators with realistic workloads. Therefore, we need a different approach.

In order to evaluate DCB in a reasonable amount of time while emphasizing on thread interactions, we use a dynamic binary translation (DBT) based emulation platform. For emulating diverse core speed for both performance asymmetry and core boosting, our plat-

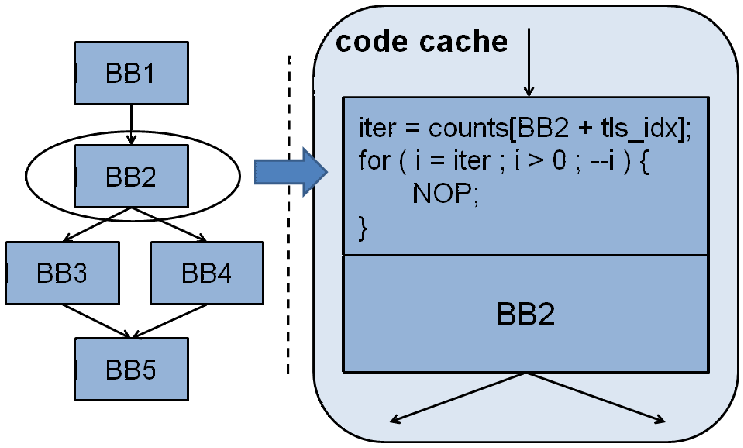

Figure 3.6: Core boosting emulation with dynamic binary translation.

form slows down execution by adding extra instructions to each basic block. Figure 3.6 shows the conceptual diagram of this scheme. The iteration counts of the inserted *nop* loop decides how much the execution is slowed down. Since we need to vary the speed from thread to thread, Thread Local Storage (TLS) is used to store the index variable tls_idx. The transition between two different core speeds can be emulated by simply overwriting the value of this variable. The counts array is loaded to the memory before executing the program.

The key point for the accuracy of this evaluation scheme is that the amount of slowdown must be inversely proportional to the modeled core speed. We achieve this by judiciously deciding the iteration counts for every basic block and for every slowdown value. Our mechanism to decide the iteration counts is inspired by Eyerman et al. [37] which states that disruptive miss events such as cache misses and branch mispredictions result in characterizable performance behavior. The basic idea is that we can accurately dictate the iteration counts according to the required slowdown amount if we can measure the per basic block number of these disruptive events.

We choose the number of instructions, the last level cache misses, and the data TLB misses, since they showed the largest correlations with the CPU time of the programs in our measurement. Using hardware performance counters, we measure these values for various time periods during repeated execution of the benchmarks. We then model the relationship between the CPU time and those variables with linear regression based on the measurement.

The hardware performance counters are also used for sampling the program counter values when the miss events occur. We collect the program counter samples to map the number of the miss events to each basic block. Assuming the sampling preserves the probabilistic distribution of the miss events, the numbers for the miss events per basic block can be calculated by projecting the sample distribution to the total number of miss events for the entire execution. Finally, the number of iterations per basic block and slowdown value are calculated according to the linear regression model along with the miss event numbers.

We have implemented the evaluation platform on DynamoRIO [13], an open source dynamic binary translation system. We perform the evaluation on a 32-core machine with four 8-core Intel Xeon processors running at 2.26GHz with 24MB L3 cache and 32GB of main memory. Except for the fact that each thread is slowed down, the execution on the evaluation platform is almost identical to running on native hardware. Since the threads actively interact with each other, the simulation errors caused by ignoring thread interactions can be minimized.

Component	Parameters
Core	4-way issue OOO
L1-I cache	32 KB, 4 way, 4 cycle, private
L1-D cache	32 KB, 8 way, 4 cycle, private
L2 cache	256 KB, 8 way, 8 cycle, private
L3 cache	8 MB, 16 way, 30 cycle, shared

Table 3.1: Simulated architecture details.

3.6.2 Evaluation of Energy Saving

We evaluate our synchronization-aware per-core power gating scheme with Sniper multicore simulator [17], version 5.2, updated with a cycle-level core model. We use Intel Nehalem [89]-like core model and the details of the simulated architecture are listed in Table 3.1. McPAT 0.8 [63] is used to estimate energy consumption.

3.7 Experimental Results

We first ascertain the validity of the evaluation platform by verifying the errors in the simulated execution time. Then, we use it to evaluate the performance improvement of DCB. We use the Pthreads implementation of PARSEC 2.1 benchmark suite [11], with *simlarge* workloads. *freqmine* is left out because it does not have a parallel version of Pthreads implementation. Although *vips* has Pthreads implementation, it is not used either since it works with GNOME Threads interface at the source code level. The current implementation of the DCB compiler needs source level interfacing with Pthreads for its static analysis. Each experiment represented is the average of the trials repeated at least ten times.

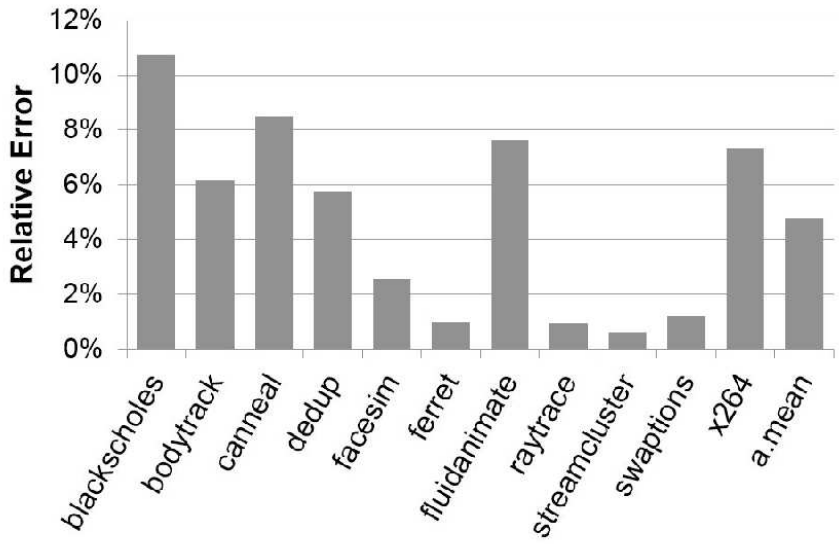

Figure 3.7: Errors in the simulated execution time of the performance asymmetry evaluation platform.

3.7.1 Accuracy of DBT-based Performance Evaluation

We verify the accuracy of our evaluation platform by comparing the execution times with slowdown. Figure 3.7 shows the errors in the simulated execution time of the platform, dropping the sign for negative values. For the experiments, we calculated the expected values from the simulated runs with 5x slowdown and compared them to the simulated runs with 10x slowdown. On average, our evaluation platform shows 4.8% of errors with the maximum of 10.8%. While our evaluation platform tries to closely match original execution using the inferred linear regression model and the per basic block hardware counter statistics, the main source of error is the difference between the original instructions and the extra instructions instrumented. Despite the fact that it does not perform the detailed microarchitectural simulation, however, it is quite accurate. More importantly, it enables us to run the programs on realistic inputs without sampling while correctly maintaining

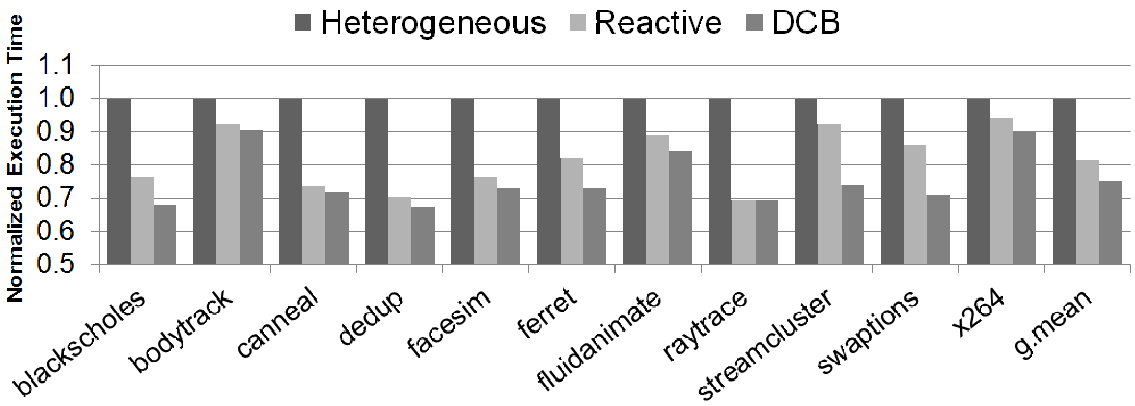

Figure 3.8: Normalized execution time of Heterogeneous, Reactive, and DCB.

inter-dependencies arising due to synchronizations.

3.7.2 DCB Performance Improvement

Using the DBT-based performance asymmetry evaluation platform, we evaluate the performance improvement of the DCB system. The underlying asymmetric CMP is assumed to be identical to the one used in Section 3.2.1. The standard deviation (σ) of the core frequencies is 30% of the average (μ), and the eight cores run at the frequencies of $(\mu - 1.5\sigma)$, $(\mu - 1.0\sigma)$, $(\mu - 0.5\sigma)$, μ, μ, $(\mu + 0.5\sigma)$, $(\mu + 1.0\sigma)$, $(\mu + 1.5\sigma)$, respectively. As the current generation of AMD processors [2] already have per-core DVFS capable of operating at 20 - 30% higher frequencies than the nominal frequencies, we use the acceleration value of 1.5x assuming fast switching ($<$ 10ns) with dual supply voltage rails. We use $c = 1$ for the boosting budget, which means one core can be boosted at any moment. We use the asymmetric CMP with no boosting, *Heterogeneous*, as a baseline. For the fairness of comparison, the frequencies of *Heterogeneous* is set to be higher than the underlying cores for the boosting schemes so that its average core frequency is equal to the boosting

schemes. Although we cannot directly measure power consumption due to the limitation of our evaluation platform, we keep the power budgets of boosting schemes as close to the baseline as possible in this way.

We also compare DCB to a reactive boosting scheme, *Reactive*, where the priority of the threads is managed in the same way as a state-of-the-art reactive core acceleration scheme, *Booster SYNC* [72]. In *Reactive*, a thread can be in one of the three priorities: *blocked*, *normal*, and *critical*. The default priority is *normal* and this changes to *blocked* when the thread is waiting for either a mutex, a condition variable, or barrier. The priority is promoted to *critical* if the thread acquires a mutex. *Reactive* always prefers the thread with higher priority. When there are multiple threads with the same highest priority, *Reactive* assigns boosting in a round robin manner.

Figure 3.8 shows the normalized execution time of *Heterogeneous*, *Reactive*, and *DCB*. *DCB* achieves performance improvement over both *Heterogeneous* and *Reactive* across all of the benchmarks. On average, the performance gain of *DCB* over *Heterogeneous* is 32.9%, outperforming *Reactive* by 10.3%. As expected from the preliminary analysis in Section 3.2.1, *DCB* is most effective for the benchmarks having thread join or barriers as the primary synchronization method, as in *blackscholes* and *streamcluster*. Interestingly, both *Reactive* and *DCB* present substantial performance improvement even for the benchmarks with dynamic workload distribution, such as *bodytrack* and *raytrace*, mainly due to the sequential regions. For the sequential portions of executions, both *Reactive* and *DCB* can concentrate the boosting budget to the only working thread yielding better performance than *Heterogeneous*.

In order to better understand the workload balancing capability of *DCB* without the

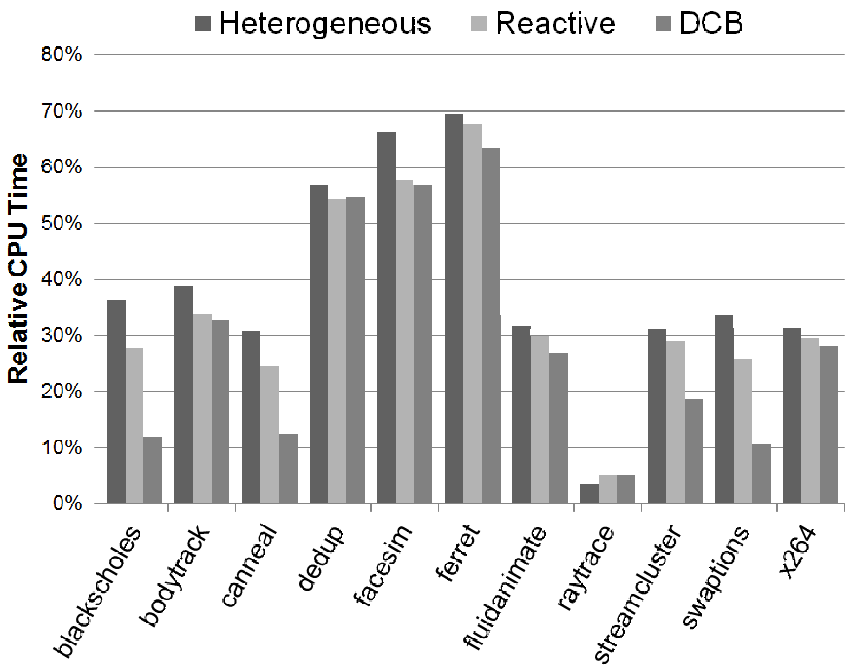

Figure 3.9: Synchronization overheads of Heterogeneous, Reactive and DCB.

effect of accelerating sequential region, we also measure the CPU time wasted for synchronization operations in the same way as in Figure 3.2. Figure 3.9 presents the CPU time portion for synchronizations. From this graph, we can confirm that *DCB* is very effective in balancing workloads and reducing the synchronization overheads, for data parallel programs such as *blackscholes* and *streamcluster*. We can also see that *DCB* can reduce the synchronization overhead of pipeline parallel programs like *ferret*. Note that this graph shows the ratio of synchronization overheads to the total CPU time of parallel execution. Since the execution time is significantly reduced for benchmarks like *dedup* and *ferret*, the workload balancing effect is actually greater than it looks in the graph.

Figure 3.10 illustrates how *DCB* outperforms the other schemes. In this figure, X-axis presents the time scale normalized against the finishing time of the last threads of *Heterogeneous*, and Y-axis is for the number of threads that have finished their tasks. Therefore, if

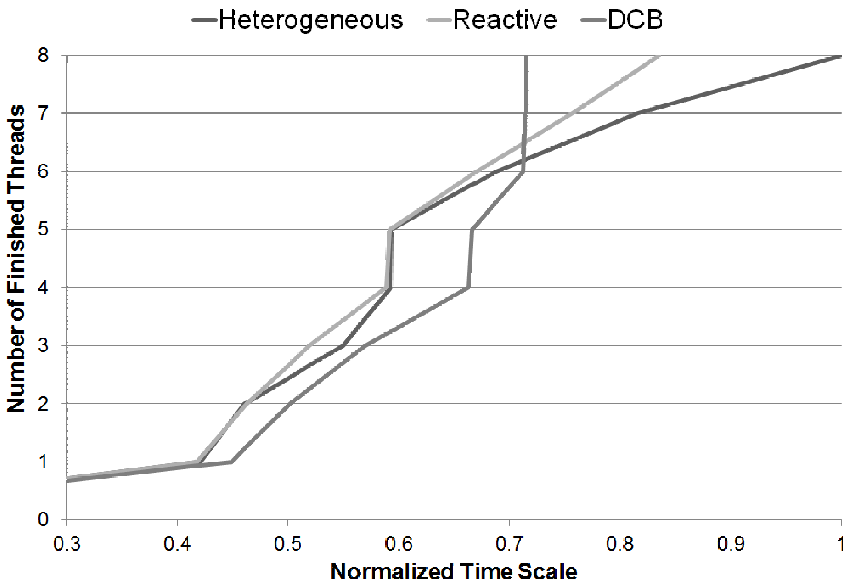

Figure 3.10: Arrival time of each thread for blackscholes.

the line hits the ceiling earlier, better performance was achieved. As expected, *DCB* shows the best performance among all the schemes. An interesting point to note is that *DCB* loses to the other schemes until the sixth thread finishes its task. This shows that *DCB* assigns the boosting budget in a way closer to the optimum. In other words, *DCB* saves the boosting budget from already fast threads and assign them to the lagging threads, reducing the workload imbalance. For this reason, the slope of the *DCB* line becomes steeper as it gets to the end. For DCB, only the last three threads are finishing their tasks approximately at the same time, and this is because the boosting budget is not enough to balance all of the threads. If *DCB* had more boosting budget, it would have the almost vertical fraction of the line from an earlier point.

Another point to notice in Figure 3.10 is that *Reactive* starts almost identically with *Heterogeneous* and gains a slightly steeper slope than *Heterogeneous*. The reason is that *Reactive* is indeed reactive. *Reactive* does not discriminate threads before some of them

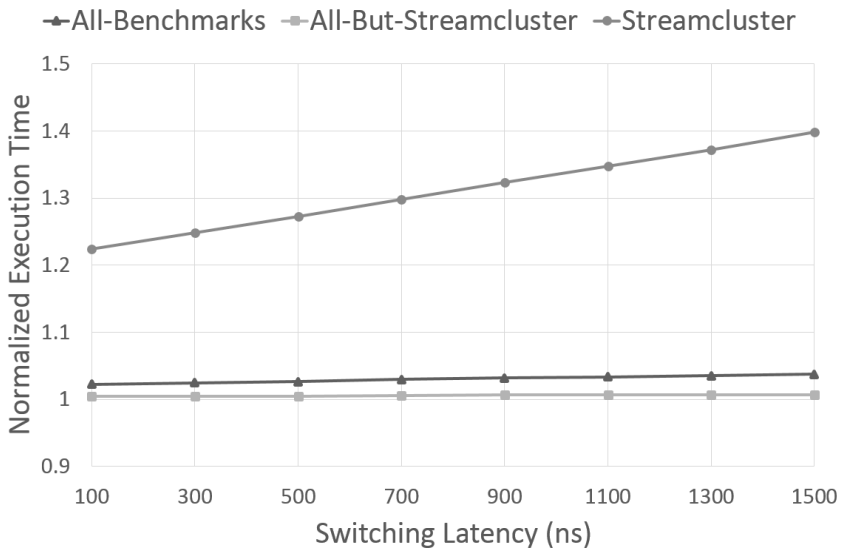

Figure 3.11: Overhead of synchronization-aware per-core power gating.

reach synchronization operations. So, it starts identical with *Heterogeneous* fairly distributing the boosting budget to all cores. This is also why *Reactive* beats *DCB* in the beginning. Moreover, the *Reactive* line is slightly steeper than *Heterogeneous* because it starts concentrating the boosting budget by not assigning it to the idle cores.

3.7.3 Energy Saving of Synchronization-Aware Power Gating

We evaluate the energy saving of our per-core power gating scheme. The underlying asymmetric CMP is assumed to be identical to the one used in Section 3.2.1. The standard deviation (σ) of the core frequencies is 30% of the average (μ), and the eight cores run at the frequencies of $(\mu - 1.5\sigma)$, $(\mu - 1.0\sigma)$, $(\mu - 0.5\sigma)$, μ, μ, $(\mu + 0.5\sigma)$, $(\mu + 1.0\sigma)$, $(\mu + 1.5\sigma)$, respectively.

Figure 3.11 shows the normalized execution time of PARSEC benchmarks when we naïvely apply per-core power gating for all synchronization stalls. For sensitivity study, we

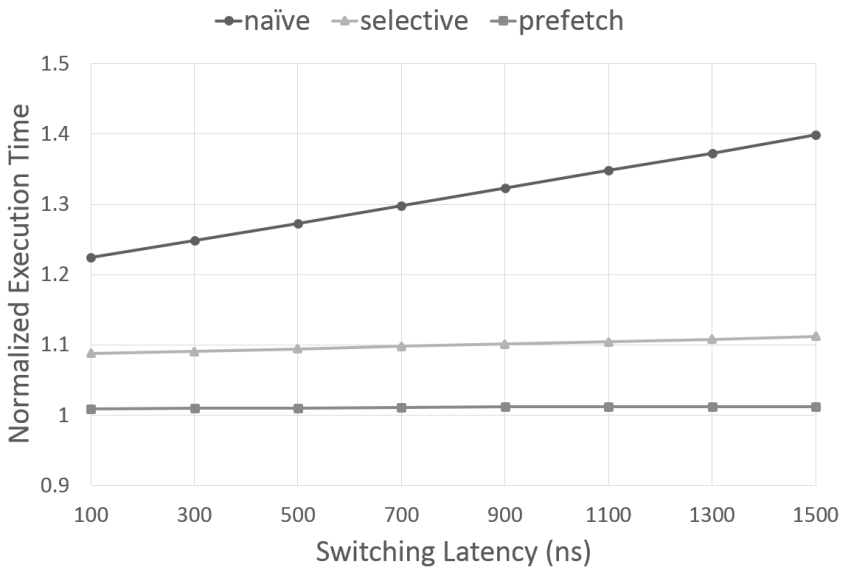

Figure 3.12: Impact of optimization for streamcluster.

vary the wake up latency from 100ns to 1500ns. Leverich et al. [60] shows 190ns of wake up latency using Spice simulation with similar setting. In this figure, we can see that power gating the idle cores waiting for synchronizations incurs negligible overhead even with the conservative wake up latency value of 1500ns, except one benchmark: *streamcluster*. This is because the frequency of synchronization stalls is relatively small compared to the duration of the stalls.

Figure 3.12 presents the impact of selective power gating and prefetching. Profiling-based selective per-core power gating reduces the number of synchronization stalls so that the longer wake up latency shows less impact on the performance. However, cold private cache after gating still incurs substantial amout of overhead, resulting in about 10% slow-down. Finally, prefetching private cache contents further reduces the overhead to about 1%.

The energy savings of the synchronization-aware power gating scheme is depicted in

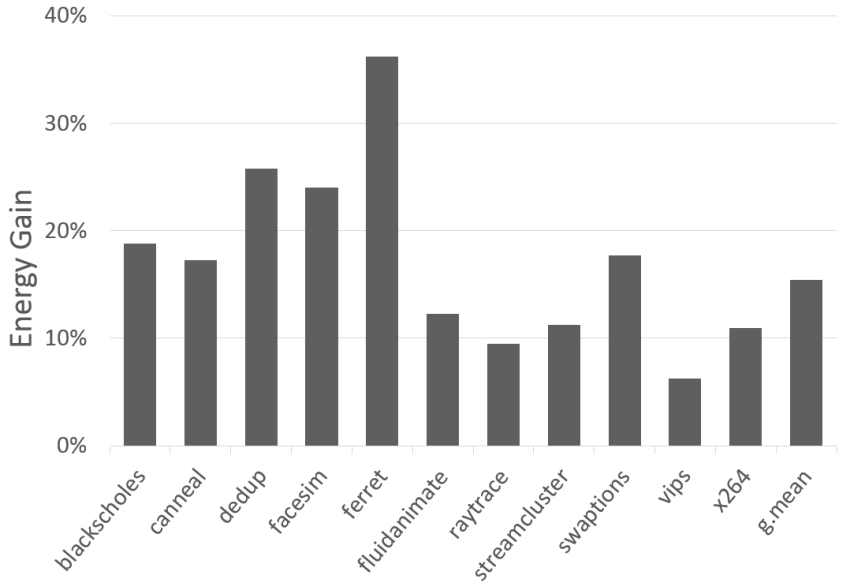

Figure 3.13: Energy savings of synchronization-aware per-core power gating.

Figure 3.13. The baseline keeps the cores in active status for synchronization blocking durations. Although the recent microprocessors support dynamic power management, they do not place the cores in sleep mode for synchronizations because the time granularities are too coarse (on the order of 100ms [60]). We assumed 300ns of wake up latency and 200nJ of switching energy overhead following the Spice simulation results of Leverich et al. [60]. Overall, our scheme improves the energy efficiency by 15% on average with only negligible performance degradation.

3.8 Related Work

In this section, we first survey previous work that suggests performance asymmetry in CMPs. Since DCB is not limited to one type of core boosting mechanism as mentioned before, we then review per-core performance adaptation technologies that could possibly be used for core boosting. Finally, we study the previous proposals for assessing thread

criticality and differentiate DCB from them.

3.8.1 Performance Asymmetry in CMPs

There have been numerous prior works that motivate inherent performance asymmetry in CMP designs. Several of them [7, 58] are proposed for better performance, and some others [57] show asymmetry is beneficial to reduce power consumption. Asymmetric CMPs have been demonstrated to be effective for alleviating serial bottlenecks [43, 91, 55]. In consequence, some commercial products [40] have started adopting the trend.

Increasing within-die process variation in near-future technologies also demands performance asymmetry even in homogeneous CMP designs. Because of process variation, Teodorescu et al. [92] claims that it is no longer accurate to think of large CMPs as homogeneous systems. Furthermore, low voltage chips aggravate the impact of process variation, and maintaining homogeneity by operating at the frequency of the slowest core severely lowers performance [71].

3.8.2 Dynamic Adaptation of Core Performance

Dynamic voltage and frequency scaling (DVFS) is a widely used technique for dynamic per-core performance adaptation [49, 36] and some AMD commercial processors support per-core DVFS [2]. However, off-chip regulator-based DVFS incurs intolerable scaling overheads (tens of microseconds) for our purpose. On the other hand, DVFS using on-chip regulators has much shorter transition time but suffers from low efficiency of the regulators.

Miller et al. [72] and Dreslinski [30] recently proposed the use of dual-voltage rails for

fast adaptation of per-core performance. In addition, Dreslinski et al. [31] confirmed that very short transition time ($<$ 10ns) is achievable with a new circuit technique. We assume this technique to demonstrate the effectiveness of the DCB system.

Another feasible option for the underlying mechanism of core boosting is adapting hardware resources of cores. Composite Cores [68] integrates two different types of computing engines and achieves high performance and energy efficiency. It also shows that fine-grained (quantum length of 1000 instructions) dynamic per-core performance adaptation is possible.

While Composite Cores adapts in-core hardware resources, Illusionist [4] uses another core to boost cores. Illusionist consists of many lightweight cores and a small number of aggressive cores, and aggressive cores are used to accelerate the execution of the lightweight cores by providing execution hints, running ahead of them.

3.8.3 Thread Criticality Assessment

Thread Criticality Predictor (TCP) [10] identifies thread criticality based on memory hierarchy statistics using hardware counters. It increases energy efficiency by scaling down the frequency of non-critical threads or improve performance by task stealing from critical threads. Although TCP shows high accuracy (average of 93%), it is not suitable for our purpose of balancing workloads for asymmetric CMPs. For example, consider two perfectly identical (including cache misses) threads running on two cores with different frequencies. In the middle of the workloads, TCP would assess the criticality of faster thread higher than the slower thread because the faster thread would have more misses to the point.

Prior work has suggested using barrier synchronizations for thread criticality prediction for saving energy either by transitioning into low power modes after reaching a barrier or by scaling down the voltage and frequency of non-critical threads. Liu et al. [64] and Thrifty Barrier [62] differ from our work as they try to predict the arrival time to the next barrier based on history while DCB only needs to decide lagging threads for data parallel programs. Meeting Points [15] is similar to our work considering that it employs instrumenting programs with special instructions for monitoring progress. However, it only works for regular parallel loops with identical iteration counts across all threads, as opposed to DCB which can handle not only the loops with varying iteration counts but also the threads with different code.

Accelerating Critical Sections (ACS) [91] and Bottleneck Identification and Scheduling (BIS) [55] also use special instructions for detecting bottlenecks. Especially, BIS measures the number of cycles spent by threads waiting for each bottleneck and accelerates the bottlenecks responsible for the highest thread waiting cycles. The primary difference of ACS and BIS from our work is that they work in coarser granularity since they rely on thread migration to accelerate bottlenecks.

The most closely related work to DCB is Booster [72], where it also tries to balance multi-threaded workloads using core boosting. They propose two boosting algorithms: Booster VAR and Booster SYNC. Booster VAR balances the CPU cycles spent by each thread and Booster SYNC improves it by taking priority hints from synchronizations. The most important difference between Booster and DCB is that Booster is reactive. Even Booster SYNC does not discriminate threads until they reach synchronization operations. Therefore, it cannot address implicit software heterogeneity caused by control flow diver-

gence and non-deterministic memory latencies. Similarly, it is not well-suited for pipeline parallel programs. Even though different stages are heavily biased, Booster gives up the chance of balancing them until some of them get blocked for synchronizations. Conversely, DCB is proactive handling software heterogeneity very well. Finally, it is not trivial to extend Booster for other types of asymmetric CMPs or core boosting mechanisms, since it uses the core frequency values for balancing cycles. Meanwhile, DCB is applicable to them without any modification for data parallel programs and it only needs relative acceleration ratio for pipeline parallel programs.

3.9 Summary

This chapter explored improving performance and energy efficiency for performance asymmetric CMPs. We investigated the elimination of workload imbalances by relying on the hardware capability to accelerate individual cores at a fine granularity. We proposed Dynamic Core Boosting (DCB), a software-hardware cooperative system that balances the workloads by boosting critical threads. DCB coordinates its compiler, runtime, and processor cores, for near-optimal assignment of core boosting. The DCB compiler instruments target programs with instructions to give progress hints. The DCB runtime subsystem monitors their execution, enabling intelligent assignment of the boosting budget for better performance. On a simulated eight core system of varying frequency, our experiments using PARSEC benchmarks showed that DCB improves the overall performance by an average of 33%, outperforming a reactive boosting scheme by an average of 10%. We also suggested applying per-core power gating to the idle cores due to workload imbalances. Our

synchronization-aware power gating scheme minimizes the performance impact of per-core power gating through selective gating and prefetching. Our scheme improves the energy efficiency by an average of 15%.

CHAPTER 4

Instrumentation Sampling for Lightweight Profiling

4.1 Introduction

As cloud computing continues to expand, profile-guided optimization (PGO) on datacenter applications has the potential for huge cost savings. Single-digit performance gains from the compiler can yield tens of millions of dollors in savings. Isolating the execution of datacenter applications can be complex or even impossible. One challenge of PGO on datacenter applications is collecting profile data from the applications running on live traffic [83]. In order to monitor production runs, the profiling overhead in terms of both throughput and latency should be kept minimal for several reasons. First and foremost, datacenter application owners are not tolerant of latency degradations (even at the 99th percentile) of more than a few percent, unlike high performance computing or other throughput-oriented applications, because they hurt the quality of service. Second, excessive profiling overhead can cause observer distortion that thwarts meaningful analysis. Finally, profiling overhead might offset the cost savings gained with PGO.

One way to keep the profiling overhead minimal is to exploit hardware support. For instance, specialized profiling hardware such as Merten's hot spot identification [70], Vaswani's programmable hardware path profiler [94], and Conte's profile buffer [26] has been proposed for low overhead profiling. Furthermore, many recent microprocessor designs have included on-chip performance monitoring units (PMU) [46, 47, 48] containing configurable performance counters that can trigger software interrupts for sampling. Google-Wide Profiling (GWP) [83] has shown that PMU-based profiling mechanisms can maintain small enough overhead to be deployed for large datacenters monitoring applications running on live traffic.

Although hardware profiling mechanisms incur low overhead, they suffer from limitations. First, the possible types of profile data are inherently defined by the features that the underlying microprocessor supports; thus, hardware profiling mechanisms are not as flexible as software-only mechanisms. In addition, PMU features are often very processor-specific, making profiling tools not portable. Lastly, as the top design priorities are hardware validation and processor performance, performance monitoring hardware tends to be considered as a second class feature with the increasing time-to-market pressures [90].

Such limitations of hardware profiling can significantly limit the potential of PGO for datacenter applications, since PGO systems must be aware of both what and how to optimize for effective optimization. Although PMUs implemented in recent microprocessors so far provide quite rich information on where to focus optimization efforts, deciding how to optimize is a considerably harder problem. For example, sampling the program counter (PC) at a high rate yields enough information to detect hot code, and current PMUs are even capable of giving finer information such as cache miss and branch mispredict PCs.

However, PMU features so far give less attention on how to optimize.

While instrumentation-based profiling mechanisms can provide more useful information about how to optimize the target applications, they tend to impose higher overheads than hardware-based mechanisms. For instance, path profiling [8] is well-known to be effective for improving code layout and superblock formation, but incurs 30-40% overhead. Other techniques such as value profiling [16] and data stream profiling [21] not only achieve gains of over 20% but also cause ten- or hundred-times slowdowns during profiling. Such high overheads prevent these mechanisms from consideration for profiling even loadtests for datacenter applications.

In this work, we propose a novel instrumentation sampling technique, *instant profiling*, that uses dynamic binary translation. Instead of instrumenting the entire execution, instant profiling periodically interleaves native execution and instrumented execution. By adjusting profiling duration and frequency parameters, we can keep profiling overhead under a few percent, so that the framework can be used to continuously monitor cloud computing applications running in large scale datacenters with live traffic. We have implemented the prototype framework of instant profiling on top of DynamoRIO [13], and we evaluate the possibility of continuous profiling on real datacenter benchmarks.

Instant profiling offers the following features:

- Low computational overhead. Computational overhead includes the cycles consumed by the application as well as out-of-band computation like profiling and JITing. When target programs are running natively, instant profiling does not need to add any extra instructions to the programs, as opposed to previous techniques [6, 44]

which need checking code even when not profiling. Also, we do not duplicate the original execution, unlike other prior work [73, 97]. For these reasons, instant profiling can keep the computational overhead minimized.

- Small latency degradation. Due to the overhead amortizing characteristics of dynamic translation techniques, end users might observe significant latency degradation for initial profiling phases even with low sampling rates. Instant profiling further reduces latency degradation by pre-populating a software code cache and jumping back to native after a predefined period.

- Eventual profiling accuracy. With sampling techniques, we cannot avoid making errors on profile data. Since our low overhead framework enables continuous profiling on production runs, however, the accuracy of instant profiling gets closer to full profiling with a long enough application lifetime or enough instances. Since the most important applications consume the most cycles, they will have the most instances, run the longest, and yield the most profiles.

- Flexibility. Instant profiling can be applied to any type of profiling or tracing as long as the entire execution does not need to be monitored, since it is an instrumentation-based profiling technique and does not rely on specialized hardware features. In addition, instant profiling is portable to other micro-architectures for the same reason.

- Tuning. The profiling duration and frequency are configurable, making it easy to adjust the tradeoff between information and overhead.

The remainder of this chapter is organized as follows. Section 4.2 provides a brief

explanation of dynamic instrumentation systems and DynamoRIO which we harness as a base platform. Section 4.3 then presents the design and implementation details of our instant profiling framework. Section 4.4 describes how the framework further reduces latency degradation by pre-populating its software code cache. Section 4.5 explores tuning trade-offs and evaluates performance. Section 4.6 discusses related work. Finally, we summarize the contributions and conclude in Section 4.7.

4.2 Background

Before we delve into the details of instant profiling, we briefly describe dynamic binary instrumentation techniques and where extra overheads come from. Then we provide an overview of DynamoRIO upon which we implement the prototype framework of instant profiling.

4.2.1 Dynamic Binary Instrumentation

Dynamic binary instrumentation is a powerful technique for runtime program introspection, particularly collecting profile data for PGO. There are many dynamic binary instrumentation systems [13, 67, 78], sharing similar internal mechanisms. They intercept target applications' execution, instrument points of interest, place instrumented code in their software code cache, and execute it from the software code cache. Where and what to instrument are defined by users (client writers) via custom API's. One main benefit of instrumenting programs at runtime is the availability of a complete picture of programs' runtime behavior including shared libraries, plugins, and dynamically-generated code.

There are two major sources of overhead for dynamic binary instrumentation systems. One arises from the dynamic instrumentation systems themselves. Whenever the target program meets an unknown branch target, the dynamic instrumentation system must perform a code cache lookup, copy the original code to the software code cache and insert any necessary instrumentation. In order to make this process transparent to target programs, moreover, they have to save and restore program context. Although these costs are unavoidable, translation overheads can be amortized over long running time and there have been suggested many optimization techniques to reduce this type of overhead, e.g., direct and indirect branch linking, trace construction, register reallocation, etc.

The other source of overhead comes from the profiling client. For collecting profile data, dynamic instrumentation systems insert user-defined code into application code. As opposed to instrumentation overhead occurring only when new code comes into the software code cache, instrumented client code is executed every time the application code is executed. Thus, even fine-tuned profiling clients can impose large overheads, continuously throughout the target application's execution. Furthermore, while significant progress has been made in reducing the performance penalty of the dynamic instrumentation itself, less attention has been paid to user-defined profiling clients [103].

4.2.2 Overview of DynamoRIO

DynamoRIO [12, 13, 1] is an open source dynamic binary instrumentation system. DynamoRIO exports an interface for building a wide variety of dynamic tools (DynamoRIO clients) including program analysis, profiling, instrumentation, optimization, etc. It allows

not just insertion of callouts/trampolines, but also arbitrary modifications to application instructions via a powerful instruction manipulation library and adaptive intermediate representation. DynamoRIO provides efficient, transparent, and comprehensive manipulation of an unmodified application running on stock operating systems (Windows and Linux) and commodity hardware (IA-32 and AMD64).

A thorough description of the internal design and implementation of DynamoRIO is outside the scope of this chapter, but is described by Bruening [13].

4.3 Instrumentation Sampling

We modify DynamoRIO's control transfer for instrumentation sampling by interleaving native execution and instrumented execution. Unmodified, DynamoRIO initially takes over the control from native execution when DynamoRIO's shared library is loaded into the target program's address space, and never gives it back. On the other hand, our instant profiling framework gives back the control to native program execution right after initialization. During initialization, it sets up a signal handler for pre-defined profiling start/stop signals and creates a shepherding thread. After it starts executing the target program natively, the framework periodically takes over and gives back the control from and to native execution for sampling.

Figure 4.1 shows how control is transfered between native execution and instrumented execution in the instant profiling framework. The shepherding thread manages control transfers by periodically sending a profiling start/stop signal to each application thread, according to the profiling duration and frequency parameters. Then, the registered signal

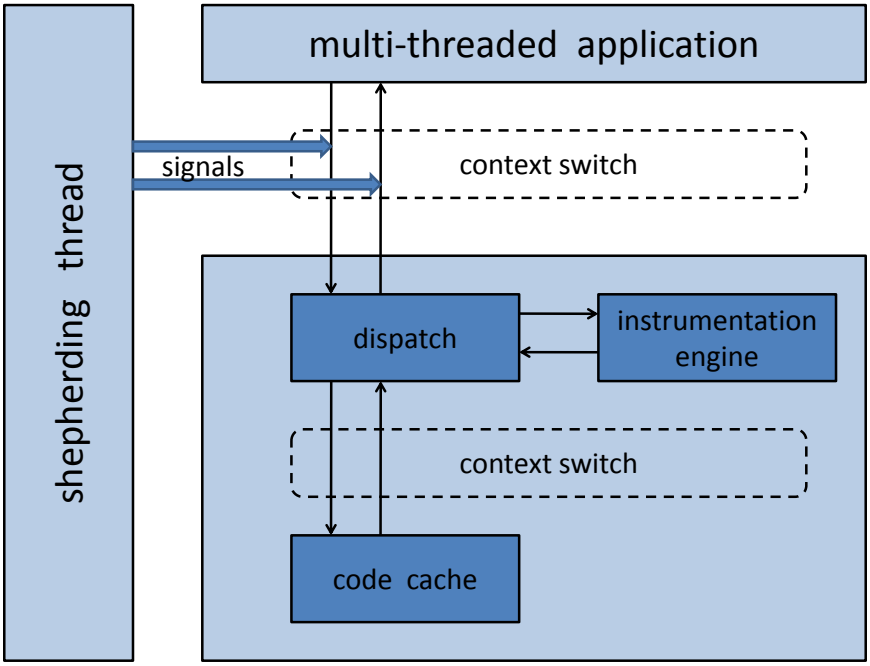

Figure 4.1: Control transfer for instrumentation sampling.

handler for the predefined signal transitions between native execution and instrumented execution. In order to make the instrumentation transparent to the target program, instant profiling needs to save and restore the target program's state every time the control is transfered via context switch.

If the profiling start signal is delivered when the thread is running natively, the signal handler saves the program state and hands over the control to the dispatch unit. iThe dispatch unit then checks whether the current program counter (PC) exists in the software code cache. If so, it restores the saved program state and executes the target code from the software code cache. If the current PC does not exist in the software code cache it invokes the instrumentation engine to instrument the target code region and place it in the software code cache. Then the dispatch unit switches the context to the software code cache.

The transition from instrumented execution to native execution happens in a similar way. In this case, however, the context switch can only occur in between two instrumented

fragments. A fragment is DynamoRIO's unit of translation and it can be either a basic block or a trace. Mapping the code cache state back to a native state is most easily done at the start or end of a code fragment. Thus, the signal handler delays the context switch until the current fragment in the software code cache finishes.

The rest of this section describes the technical issues involved in making the start/stop profiling transitions lightweight and transparent to the target program.

4.3.1 Context Switch

For a sampling mechanism to be effective, transitions of start/stop profiling should be very lightweight. Otherwise, the transitions would encroach on the overhead budget. In order to make the transitions lightweight, our instant profiling framework minimizes the operations needed for the context switch between native execution and instrumented execution.

The framework performs a context switch to start profiling as follows: when the start profiling signal handler gets a signal, the kernel hands over the machine context of native execution to the signal handler in the form of a *sigcontext* struct, which the handler passes to the dispatch unit after modifying a few fields To invoke the dispatch unit, the ip register in the sigcontext is set to the re-entry point of the dispatch unit. Then, when the signal handler returns, the kernel gives the control to the dispatch unit. The dispatch unit starts instrumented execution starting from the program counter value saved from the sigcontext struct.

4.3.2 Temporal Unlinking and Relinking of Fragments

One of DynamoRIO's optimizations that has a large impact on performance is direct and indirect branch linking. Since a context switch between the software code cache and the dispatch unit is expensive, DynamoRIO links branches instead of switching context whenever a branch target exists in the software code cache.

Although it is good for performance, the direct and indirect branch linking optimization can cause a problem for sampling control. Assume a thread is running inside a loop linked in the software code cache. When the stop profiling signal is delivered, the signal handler sets up the control transfer and continues running in the software code cache since it is in the middle of a fragment. In this case, however, the control transfer does not happen until the execution actually finishes running the loop and returns to the dispatch unit. For this reason, the direct and indirect branch linking optimization can cause unbounded profiling.

In order to prevent unbounded profiling, our instant profiling framework temporarily unlinks the outgoing branches of the currently running fragment when it gets the stop profiling signal. For better performance, the framework needs to re-link the branches afterward. So, it saves the unlinked branches in a scratch-pad data structure and re-links them when it restarts profiling.

4.3.3 Multi-threaded Programs

Although unmodified DynamoRIO seamlessly supports multi-threaded programs, we need several special treatments due to the structural difference between our instant profiling framework and DynamoRIO. The key issue is how to take over the control of all threads

when we want to start profiling. This is not a problem for unmodified DynamoRIO since it takes over the control of the main thread before it spawns any other threads, observes every system call including thread creation, and never gives up the control of any thread. On the other hand, our framework only takes over the control when it is doing profiling, and does not keep supervision when the threads are running natively.

The basic strategy that our framework takes is to force its own signal handler for every thread and to send a profiling start signal to each thread. The shepherding thread can enumerate the thread IDs of every application thread even when they are not created and/or running under control of the framework, and send each thread a pre-defined signal that can be easily configured with a parameter. Since the kernel calls the registered signal handler when the signal is delivered, the framework can take over the control of every thread whenever it needs to in this way.

One problematic case is when the target program tries to mask the signal that we use or to register another handler for the signal. In this kind of conflict, the simplest circumvention is to use a different signal that is not touched by the application. For this purpose, the signal number we use as the start/stop profiling signal can be easily configured via a command-line parameter. Another solution is to intercept those tries by slipping in our wrapper functions for the library functions such as sigaction(), signal(), or sigprocmask(). In this case there still can be applications which directly call system calls (e.g., with assembly language), and they need to be handled with ptrace. They are extremely rare cases, however, especially for datacenter applications which mostly use standard libraries for portability. Finally, for the programs we have tested so far, changing the signal was enough.

4.3.4 Summarizing Profile Data

In order to enable profiling clients to summarize their results, our instant profiling framework extends DynamoRIO's API. DynamoRIO has various API functions to register customized instrumentation points and we add one more type of such event.

- dr_register_profiling_end_event(function)

The function registered with this API is called by the shepherding thread after every profiling phase. In this way profiling clients can manage profile data. The summarizing overhead can be hidden as it is performed in the shepherding thread and not included in an application's critical path.

4.4 Pre-populating Software Code Cache

Our instant profiling framework further reduces the latency degradation by pre-populating its software code cache. As mentioned in Section 4.1, minimizing latency degradation is extremely important for datacenter applications as it is directly related to the applications' quality of service. Many systems have expected 99th percentile latencies under 10ms. Meanwhile, using a software code cache technique amortizes its translation overhead over continued reuse of translated code. This means that end users may observe latency degradation for initial profiling phases even though we keep average overhead very small by setting a low profiling frequency. Instant profiling does not have to manifest instrumentation overhead to users, however, as it does not always run the programs from the software code cache. In other words, we can hide instrumentation overhead by instrumenting target code in parallel while the program is running natively. This can be understood in a similar way

to prefetching into an instruction cache implemented in many modern micro-architectures, and we call this technique pre-populating a software code cache.

Our instant profiling framework decides which code regions to instrument for pre-populating its software code cache based on locality. When the target program is running natively, it uses hardware performance monitoring units to collect program counter samples. It is likely that those code regions with high sample counts will be executed again when the framework starts profiling. Therefore, it pre-populates its software code cache with the basic blocks containing the program counter whose counts counts exceed a threshold.

4.4.1 Finding Basic Block Headers

Finding code regions to instrument from program counter samples is not a trivial task, especially for processors with variable-length instructions like IA-32/AMD64. For DynamoRIO, code fragments are tagged and managed with the program counter values of their first instructions. Given a program counter, therefore, we need a mechanism to find the basic block header including that program counter.

One heuristic can be backward decoding. Starting from the target program counter, it decodes previous bytes until a valid instruction is found. The heuristic repeats this process until it meets a branch instruction, at which point it takes the post-branch program counter as the basic block header. With RISC architectures where instructions have fixed length, backward decoding works quite well. However, the overhead is too high for architectures with variable-length instructions. The overhead prohibits it from being used for datacenter

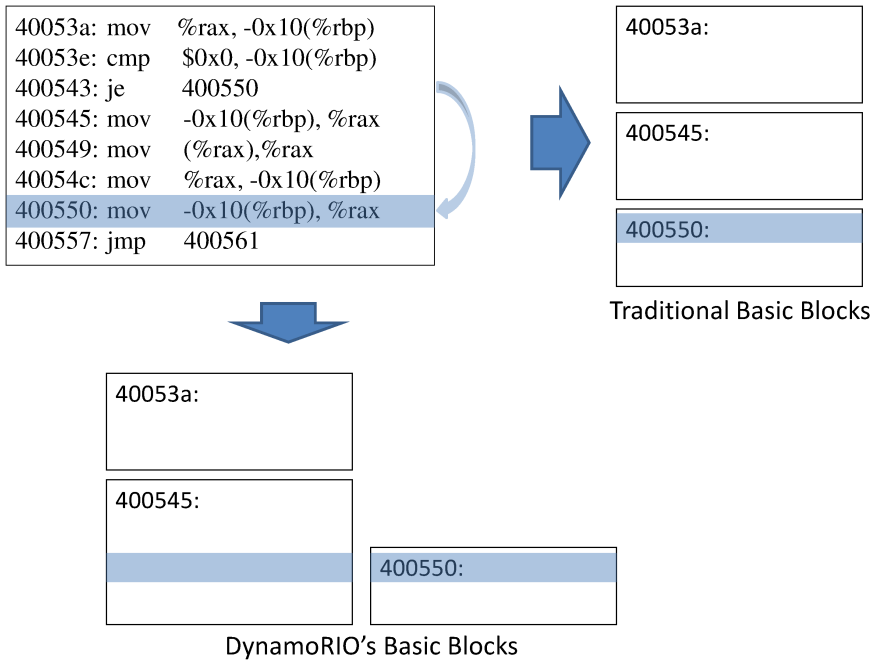

Figure 4.2: Traditional vs. DynamoRIO's basic blocks.

applications, since our framework works on IA-32/AMD64 processors.

Instead of the backward decoding heuristic, our framework performs forward decoding. From the entry points of text segments, it decodes consecutive instructions in order and also records branch targets. After finishing this process, instructions following branch instructions and branch targets start new basic blocks. We save these basic block header addresses in sorted order. Then, we can identify the basic block header containing a given program counter with binary search. The overhead of initial basic block header calculation can be hidden by performing it before the start of profiling, or it can be done offline.

4.4.2 Affinity-based Pre-population

A given program counter sample can be in multiple basic blocks for DynamoRIO since its basic blocks are different from the traditional static analysis notion of basic blocks. The example in Figure 4.2 shows the difference between traditional basic blocks and Dy-

namoRIO's basic blocks. DynamoRIO considers each entry point to begin a new basic block, and follows it until a control transfer is reached, even if it duplicates the tail of an existing basic block.

DynamoRIO uses this notion for simplicity of code discovery at runtime [13], but it can decrease the hit ratio of software code cache pre-population. For instance, suppose program counter 400550 in Figure 4.2 is sampled for pre-population. The basic block header found by the search in Section 4.4.1 will yield only 400550. For actual instrumented execution, however, both basic blocks starting from 400545 and 400550 can be encountered.

In order to solve this problem and exploit spatial locality in higher degree, our instant profiling framework adopts affinity-based pre-population. Instead of just pre-populating the software code cache with basic blocks containing sampled program counter, the framework also instruments additional basic blocks close to those basic blocks. Starting from the basic blocks found from program counter samples, it includes the branch targets of those basic blocks. It discovers target basic blocks in a breadth-first-search-like manner to a pre-defined depth.

4.5 Performance Evaluation

Instant profiling balances a tradeoff between information and overhead. This balance can be controlled with two parameters. The first parameter, profiling duration, controls how long one profiling phase lasts. A longer profiling duration gives more information, but also incurs higher overhead. Moreover, it is possible that end users might feel intermittent latency degradation during profiling phases. So we limit profiling durations to a few mil-

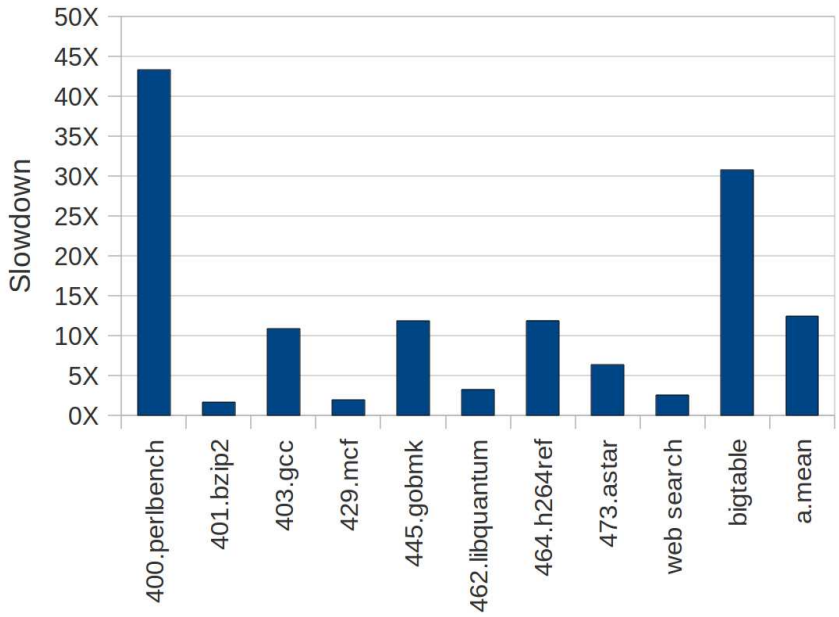

Figure 4.3: Overhead of edge profiling.

liseconds at maximum. Another parameter that affects the profiling overhead is profiling frequency. Considering datacenter applications' long running characteristics, our scheme of profiling a very small portion of execution can yield arbitrarily low average computational overhead, while still giving meaningful profile data. Since most of our benchmark workloads run only for a few tens to hundreds of seconds, however, we set profiling frequency relatively high – once in a few seconds at minimum. In these experiments, a pair of profiling duration and frequency parameters sets how long and how often profiling is performed. For example, the (2ms/4s) setting means profiling is conducted for 2 milliseconds for every 4 seconds. We compare results for (2ms/4s), (1ms/1s), (2ms/1s), (4ms/1s), and (2ms/250ms).

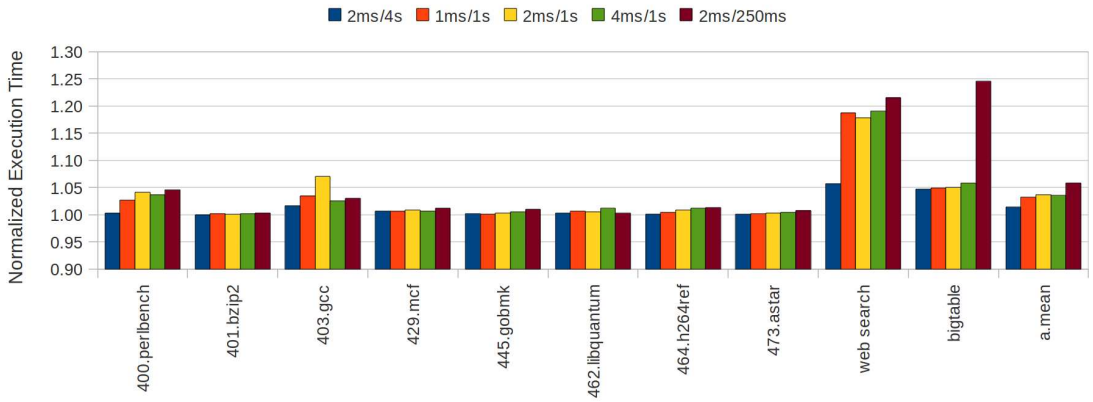

Figure 4.4: Execution time overhead of the instant profiling framework across five configurations of *(duration / frequency)*.

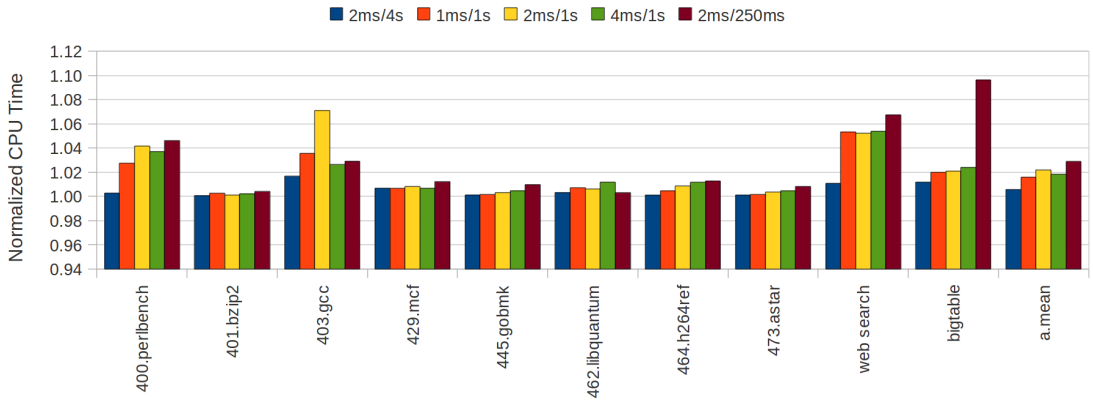

Figure 4.5: Computational overhead of the instant profiling framework across five configurations of *(duration / frequency)*.

4.5.1 Experimental Configuration

All experiments are performed on a system with a 6-core Intel Xeon 2.67GHz processor with 12,288KB L3 cache. The system has 12GB of memory and is running Linux kernel version 2.6.32. We used gcc 4.4.3 to compile all binaries with -O3 optimization.

Instant profiling is evaluated using the SPEC CPU2006 integer benchmark suite and two proprietary datacenter application benchmarks. For the SPEC CPU2006 benchmark suite, the floating point benchmarks are omitted because they generally exhibit highly repetitive behavior that is not as interesting from the perspective of profiling. In addition, four integer

benchmarks are omitted because our prototype framework does not yet work for them. The datacenter applications are web search and BigTable [18]. Although each experiment presented is the average of three repeated trials, there still exists some degree of variability in performance and accuracy due to the non-determinism caused by random starting points of profiling and thread interleaving.

4.5.2 Edge Profiling

We choose edge profiling as a profiling client to demonstrate the effectiveness of instant profiling, since it is widely used and relatively simple to implement, but incurs considerable overhead. Edge profiling is a traditional control flow profiling technique for profile-guided optimization. It measures how many times each edge (branch transition) in control flow graphs executes, and has been the basis of path-based optimizations that select hot paths. Although edge profiling collects strictly less information than path profiling, Ball[9] shows that various hot path selection algorithms based on edge profiles work extremely well in most cases.

Figure 4.3 presents the overhead of our edge profiling client, when it runs on original DynamoRIO without sampling. This naive implementation has little tuning or optimization, and its overheads are far larger than other optimized edge profiling techniques [33]. Although there are opportunities for optimizing the client itself, it is outside the scope of this chapter and we demonstrate the effectiveness of instant profiling by showing how it performs even with a naively implemented experimental client. Since the tradeoff between information and overhead is tunable with sampling parameters, edge profiling makes a

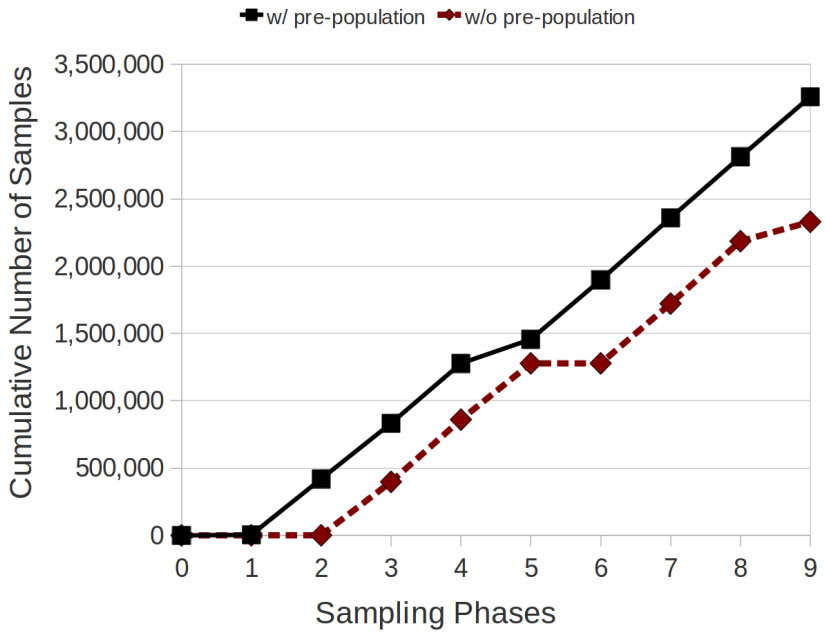

Figure 4.6: Effect of pre-populating a software code cache.

good test case because comparing edge profiles' quality is well studied.

4.5.3 Performance Overhead

The slowdowns caused by our instant profiling framework with the edge profiling client are shown in Figure 4.4. They are calculated as the profiled execution time (wall time) divided by the native execution time. Figure 4.5 also shows the computational overheads, which is calculated with CPU time. For all configurations tested, the average slowdown ranges from 1.4% to 5.9%, and the average computational overhead ranges from 0.6% to 2.9%.

The main trend that can be observed is that increasing sampling rate either by increasing profiling duration or profiling frequency results in an increase in overhead. We chose profiling frequency once in every 4 seconds at least, since a few benchmarks only run about 30

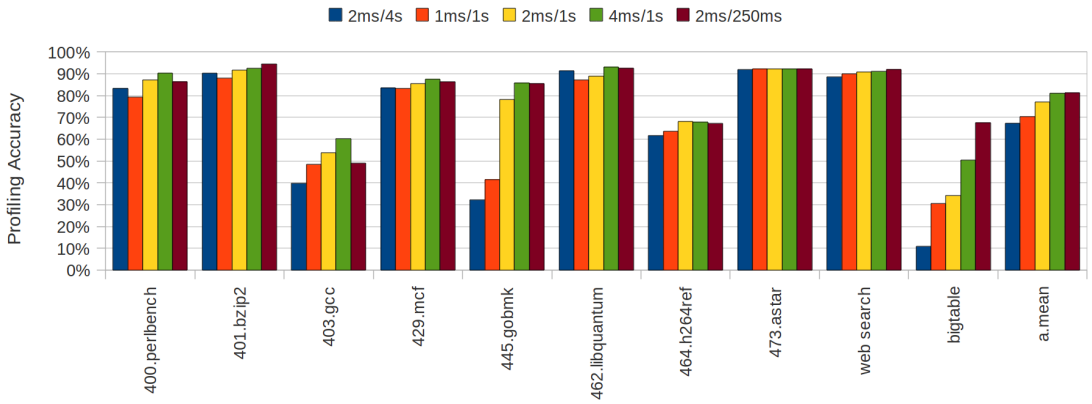

Figure 4.7: Edge profiling accuracy of the instant profiling framework across five configurations of *(duration / frequency)*.

seconds. For real datacenter environments, however, applications usually run much longer and there exist many instances of the same application running concurrently. In production environments, we can choose a much lower profiling frequency and expect commensurately lower overheads.

Although instant profiling can be tuned to impose very low average computational overhead, some of the configurations caused some benchmarks to slow down by up to 25%. There are two major locations where instant profiling adds extra instructions. One is profiling phases of every thread, but the durations of this type are controlled by the profiling duration parameter. The other location is the shepherding thread, especially the profile data summarizing phase. For the edge profiling client we used for the experiments, the shepherding thread summarizes and prints out profile data to disk after every profiling phase. While this overhead can be hidden for most benchmarks since it is not in the application's critical path, it can cause resource contention resulting in slowdowns. Although it is not yet clear, in our edge profiling case we think the resources that cause the slowdown are the data cache and load store queue. The two datacenter applications have larger working

set size than SPEC benchmarks, and our edge profiling client traverses edge counters after every profiling phase. This increases the pressure on the data cache. Also, we use atomic increment instructions to modify edge counters for the datacenter benchmarks, since they are highly multi-threaded and non-atomic increments can cause data races on the counters in this case. This can impose substantial contention on the load store queue. As we can see with the bars where profiling frequency is 4 seconds, however, even the overhead caused by the resource contention of naively implemented profiling clients can be kept small with proper parameter settings. Moreover, we expect this overhead would go further down with practical profiling frequency in real datacenter environments.

We also examine how pre-populating a software code cache can reduce latency degradation. Figure 4.6 shows the cumulative number of samples with and without pre-population, for the web search benchmark with the (4ms/1s) setting. As can be observed in the graph with small slope phases, instrumentation overhead to populate the software code cache can result in a small number of samples, and thus more latency degradation, for initial profiling phases. Pre-populating a software code cache reduces such degradation by decreasing the software code cache miss rate.

4.5.4 Profiling Accuracy

The accuracy of the edge profiling client can be ascertained by comparing the sampled profile with the profile collected with full instrumentation. We adopt a method similar to Wall's weight matching scheme [96]. We define edge profiling accuracy as

$$Accuracy = \frac{(MaxError - Error)}{MaxError} \times 100(\%) \tag{4.1}$$

$$Error = \sum_{e \in Edges} |freq_{full}(e) - freq_{sampled}(e)| \quad (4.2)$$

In the second equation, $freq_{full}(e)$ and $freq_{sampled}(e)$ represent relative frequencies of edge e in a fully instrumented profile and a sampled profile, respectively. Relative frequency is defined as the number of times that an edge is taken divided by the number of times any edge in the profile is taken. For the worst case results where edges are biased the opposite way, this error sums up to 2, defining $MaxError$ as 2.

The profiling accuracy of our instant profiling framework for edge profiling is shown in Figure 4.7. For all configurations tested, the average accuracy ranges from 67-81%, and many of the benchmarks achieve about 90% accuracy.

Despite many sources of noise, we can observe the general trend of increasing accuracy as profiling duration or profiling frequency increase. The more samples the framework collects, the closer the profile data gets to full instrumentation.

This can also be seen in Figure 4.8, which shows how the average accuracy changes as the number of profiling phases increases for two datacenter application benchmarks with (2ms/250ms) parameter setting. In the graph, although the curves are not strictly monotonic, we can see the accuracy generally goes up as more samples are collected.

Although the edge profile accuracy of our framework reaches 90% for many of the benchmarks, some benchmarks such as BigTable and gcc show very low accuracy. The main reason for the low accuracy is that our framework could not collect enough samples as the execution time of these benchmarks is too short. For real datacenter environments, however, having low overhead is paramount and it can be tuned to collect profile data that

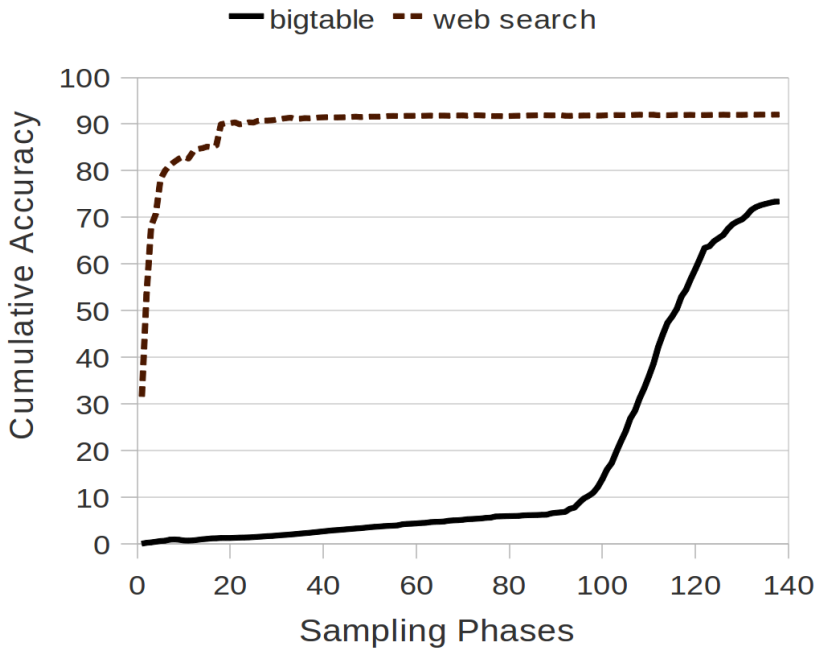

Figure 4.8: Asymptotic edge profiling accuracy.

would yield accuracy that is actionable with PGO.

4.6 Related Work

Inspired by the Digital Continuous Profiling Infrastructure (DCPI) [3], Google-Wide Profiling (GWP) [83] demonstrates continuous profiling is possible for datacenter applications running with live traffic. Although GWP also collects some lightweight callstack-based profiles through specialized libraries, it mainly relies on performance monitoring units (PMU) supported by recent microprocessors [48] to collect system-wide profiles with low overhead. The types of profiles GWP collects, therefore, are limited to the ones that either PMUs support or can be collected with specialized libraries. Our work tries to extend GWP for collecting more general profiles which can be gathered only through instrumentation. These types of profiles will enable more profile-guided optimizations (PGO) by

providing profiles that can help figure out not only ”what to optimize” but also ”how to optimize.”

Dynamic instrumentation tools such as DynamoRIO [13], Pin [67], and Valgrind [78] help instrument an application and collect general profiles of full execution. Even for simple profiles, however, the overhead of instrumenting an entire execution is prohibitive and infeasible to be deployed on datacenter applications running with live traffic.

One way to reduce the overhead of profiling is sampling, as several instrumentation approaches have demonstrated. Among them, the Arnold-Ryder instrumentation framework [6], implemented in the Jalapeno JVM, significantly lowers instrumentation overhead by sampling bursts of execution. It creates two versions of each procedure, one for checking and the other for actual profiling. The checking version counts how many times it is executed at procedure entries and loop back edges, and transitions to the profiling version if the counter reaches some pre-defined value. The profiling version collects an intra-procedural acyclic trace, resets the counter, and transitions back to the checking version. Bursty Tracing [44] extends the Arnold-Ryder framework for longer inter-procedural traces and further reduces the overhead with a few optimizations. In addition, Bursty Tracing is applied to IA32 binaries using the Vulcan binary rewriting tool, instead of Java bytecode.

Instant profiling is partially inspired by the Arnold-Ryder framework and Bursty Tracing. Instead of instrumenting all execution for checking, however, it does not instrument any code when it is not profiling. This is because even simple checking instrumentation imposes prohibitive overhead for datacenter applications. For example, even without any profiling client, the Arnold-Ryder framework results in instrumentation overhead of 6-35%, and Bursty Tracing lowers it to 3-18% [44]. On the other hand, instant profiling imposes

less than 10% of computational overhead with a naive implementation of edge profiling. Unlike Arnold-Ryder or Bursty Tracing, moreover, instant profiling will incur neither latency degradation nor computational overhead while it is not profiling. Also, instead of managing a duplicate copy of every code region, instant profiling only JITs things it will likely need. Ephemeral Instrumentation [93] also takes a similar sampling approach to Bursty Tracing, but it is non-trivial to extend Ephemeral Instrumentation for many profile types because it uses branch patching; only information available at the branch can be recorded and it is difficult to find extra registers for architectures like x86. Conversely, instant profiling is flexible and not limited to any specific profile type. Finally, phase-guided profiling techniques [86] can help sampling-based profiling methods, including instant profiling, maintain higher accuracy while keeping the overhead low.

Another vein of previous work to reduce profiling overhead is to exploit parallelism for profiling. As the micro-architectural trends move toward massively multi-core processors, Shadow Profiling [73] and SuperPin [97] aim to leverage the abundance of extra hardware. Shadow Profiling runs the original program uninstrumented in parallel with instrumented slices to perform profiling. SuperPin uses a similar approach, but tries to deterministically replicate full execution by creating slices of execution between each system call. They both exploit modern kernels' copy-on-write mechanism by forking new processes for profiling. They significantly reduce the slowdown caused by profiling since the original process is not instrumented. However, SuperPin is not deployable for datacenters as it at least doubles resource contention, especially CPU utilization and memory bandwidth. Also, virtualizing fork for multi-threaded programs is very challenging to implement robustly and their initial implementations only support single-threaded programs.

PiPA [103] also exploits parallelism but in a different way. Instead of profiling in an extra process, it performs profiling in the same thread to produce compact profiles, and uses multiple threads to pipeline processing and analyzing of profile data. PiPA is particularly effective for the types of profiling that need complicated post-processing such as cache simulation.

There have been suggested many techniques specialized for other types of profiling. Ball [8] proposes techniques for path profiling. Calder [16] suggests an optimization to turn off profiling by realizing profile data is converging for value profiling. Chilimbi [20] proposes a compact representation for memory stream profiles. Instant profiling is orthogonal to these profile-specific techniques including PiPA, and they can be used to further improve the overhead.

4.7 Summary

We introduce instant profiling, a novel approach to reduce the overhead of instrumentation-based profiling for datacenter applications. The technique works by executing instrumented profiling code from a software code cache for only a short profiling duration. For normal execution phases, the original binary runs natively without any instrumentation. We further avoid possible latency degradation for initial profiling phases by pre-populating the code cache. The prototype framework of instant profiling is built on top of DynamoRIO, and it is evaluated on the SPEC CPU 2006 integer benchmark suite and two datacenter application benchmarks. We show that the overhead of profiling in terms of both throughput and latency can be kept to acceptable levels for continuous profiling of live datacenter appli-

cations. Furthermore, we have shown that sampling-based continuous profiling can yield asymptotically accurate profiling results with negligible overhead by collecting profile data over many instances or a long time period.

CHAPTER 5

Conclusion

5.1 Summary

As power dissipation limits and design complexity have been preventing the semiconductor industry from improving the performance of monolithic processors, chip multiprocessors (CMPs) have grown into a standard to improve application performance. Since sufficient thread level parallelism (TLB) is necessary to benefit from the computational power provided by CMPs, most performance-conscious programmers face increasing pressure to parallelize their programs.

For the most prevalent parallel programming model of shared-memory multi-threaded programs, synchronization operations such as mutexes, condition variables, and barriers, play a critical role of enforcing the threads to interact with each other in the way the programmers intended. However, utilizing synchronization operations in both correct and efficient way at the same time is extremely difficult, and programmers often make trade-offs between the programmability and the efficiency of employing synchronization operations.

In this dissertation, we investigated a set of solutions that increase the programmability and efficiency of concurrent programs by intelligently managing inter-thread synchronization dependencies.

In Chapter 2, we presented practical lock/unlock pairing mechanism for C/C++ which helps concurrency bug detection tools and automated bug fixers improve correctness of concurrent programs. This mechanism combines static analysis and dynamic instrumentation to identify critical sections. It first applies a conservative inter-procedural path-sensitive dataflow analysis to pair up lock and unlocks. When the static analysis is not successful, our method makes likely assumptions based on common heuristics, and the assumptions are checked at runtime using lightweight instrumentation. With the experiments on large server programs, we proved our mechanism can pair up most locks and unlocks with small overhead incurred by the runtime check.

Chapter 3 targeted improving performance and energy efficiency of concurrent programs for performance asymmetric CMPs. Workload imbalances in asymmetric CMPs cause more CPU cycles wasted for synchronizations. We investigated the elimination of workload imbalances by relying on the hardware capability to accelerate individual cores at a fine granularity. Dynamic Core Boosting (DCB), a software-hardware cooperative system, is suggested to accelerate critical paths formed by synchronization operations by coordinating compiler, runtime, and processor architecture. In addition, we proposed applying per-core power gating to the idle cores blocked for synchronization operations. Our synchronization-aware per-core power gating scheme could significantly improve energy efficiency while minimizing the performance degradation through selective gating and hinted prefetching.

Instant Profiling, a lightweight flexible profiling mechanism that could be used for the other parts of the dissertation, is presented in Chapter 4. Instant Profiling maintains low profiling overhead by combining instrumentation sampling and dynamic binary translation. It further reduces the latency degradation by pre-populating a software code cache. We showed that meaningful profile data could be collected in production run with negligible overhead.

This dissertation has introduced a number of novel techniques to intelligently managing inter-thread synchronization dependencies. Although these techniques focused on various aspects from correctness to performance and energy efficiency, they are inspired by the same observation that neither a static nor a dynamic approach is sufficient. By applying the theme of hybrid static/dynamic mechanism to managing synchronization dependencies, we explored the possibility of increasing programmability and efficiency of concurrent programs.

5.2 Future Work

As power dissipation and thermal constraints become primary factors to be considered for microprocessor design, performance and energy efficiency cannot be pursued independently. In this dissertation, synchronization-aware dynamic core boosting and per-core power-gating are proposed for better performance and energy efficiency, respectively. We investigated them separately due to the limitations of our evaluation methodologies.

A natural direction to extend this work is to apply both of the mechanisms at the same time. In such systems, a core can be in one of the three modes: normal, boosted, and power-

gated. For simple cases, straightforwardly applying our schemes might work well, i.e. critical threads are accelerated in boosted mode while non-critical threads are power-gated to save energy. However, there could be cases where the two schemes conflict. For example, reducing workload imbalances might eliminate the opportunities to save energy with power-gating. On the other hand, they could be sometimes synergistic. For instance, power-gating idle cores could enable boosting more number of cores simultaneously. Therefore, more sophisticated assignment algorithms are necessary for applying both core boosting and per-core power-gating. Furthermore, the policy must be carefully designed depending on which attribute has higher priority.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Dynamorio: Dynamic instrumentation tool platform - http://dynamorio.org/home.html. 93

[2] AMD. *AMD Family 10h Server and Workstation Processor Power and Thermal Data Sheet*, June 2010. http://support.amd.com/us/Processor_TechDocs/43374.pdf. 51, 76, 83

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, pages 1–14, 1997. 110

[4] A. Ansari, S. Feng, S. Gupta, J. Torrellas, and S. Mahlke. Illusionist: Transforming lightweight cores into aggressive cores on demand. In *Proc. of the 19th International Symposium on High-Performance Computer Architecture*, pages 436–447, 2013. 84

[5] The Apache HTTP Server Project, 2012. `http://httpd.apache.org`. 32

[6] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *Proc. of the '01 Conference on Programming Language Design and Implementation*, pages 168–179, 2001. 90, 111

[7] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, pages 506–517, 2005. 45, 83

[8] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996. 90, 113

[9] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 134–148, 1998. 105

[10] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, pages 290–301, 2009. 84

[11] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, 2008. viii, 5, 6, 49, 74

[12] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proc. of the 2003 International Symposium on Code Generation and Optimization*, pages 265–275, 2003. 93

[13] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institude of Technology, Sept. 2004. 41, 73, 90, 92, 93, 94, 102, 111

[14] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008. 39

[15] Q. Cai, J. Gonzalez, R. Rakvic, G. Magklis, P. Chaparro, and A. Gonzalez. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *Proc. of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 240–249, 2008. 85

[16] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *The Journal of Instruction-Level Parallelism*, 1, 1999. 90, 113

[17] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proc. of the 2011 International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2011. 74

[18] F. Chang, J. Dean, S. Chemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 205–218, 2006. 105

[19] S. Cherem, L. Princehouse, and R. Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 480–491, 2007. 10, 11, 41

[20] T. M. Chilimbi. Efficient representation and abstractions for quantifying and exploiting data reference locality. In *Proc. of the '01 Conference on Programming Language Design and Implementation*, pages 191–202, 2001. 113

[21] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proc. of the '02 Conference on Programming Language Design and Implementation*, pages 199–209, 2002. 90

[22] H. K. Cho and S. Mahlke. Dynamic acceleration of multithreaded program critical paths in near-threshold systems. In *Proc. of the 2012 Workshop on Near-Threshold Computing*, 2012. 8

[23] H. K. Cho and S. Mahlke. Enbracing heterogeneity with dynamic core boosting. In *2014 Symposium on Computing Frontiers*, 2014. 8

[24] H. K. Cho, T. Moseley, R. Hank, D. Bruening, and S. Mahlke. Instant profiling: Instrumentation sampling for profiling datacenter applications. In *Proc. of the 2013 International Symposium on Code Generation and Optimization*, pages 1–10, 2013. 8

[25] H. K. Cho, Y. Wang, H. Liao, T. Kelly, S. Lafortune, and S. Mahlke. Practical lock/unlock pairing for concurrent programs. In *Proc. of the 2013 International Symposium on Code Generation and Optimization*, pages 1–12, 2013. 8

[26] T. M. Conte, B. A. Patel, and J. S. Cox. Using branch handling hardware to support profile-driven optimization. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 12–21, 1994. 89

[27] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977. 39

[28] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. of the '02 Conference on Programming Language Design and Implementation*, pages 57–68, 2002. 10, 11, 40, 41

[29] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *Proc. of the '08 Conference on Programming Language Design and Implementation*, pages 270–280, 2008. 10, 40

[30] R. G. Dreslinski. *Near Threshold Computing: From Single Core to Many-Core Energy Efficient Architectures*. PhD thesis, University of Michigan, 2011. 51, 52, 83

[31] R. G. Dreslinski, B. Giridhar, N. Pinckney, D. Blaauw, D. Sylvester, and T. Mudge. Reevaluating fast dual-voltage power rail switching circuitry, June 2012. In 10th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) in conjunction with ISCA 39. 51, 84

[32] K. Een and N. Sorensson. An extensible sat-solver [ver 1.2], 2003. 17, 22

[33] A. E. Eichenberger and S. M. Lobo. Efficient edge profiling for ilp-processors. In *Proc. of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pages 294–303, 1998. 105

[34] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proc. of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003. 9, 10, 32, 40, 41, 42

[35] S. Eyerman, K. D. Bois, and L. Eeckout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Proc. of the 2012 IEEE Symposium on Performance Analysis of Systems and Software*, pages 145–155, 2012. 2, 5, 6

[36] S. Eyerman and L. Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization*, 8(1), 2011. 51, 83

[37] S. Eyerman, L. Eeckout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems*, 27(2), 2009. 72

[38] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987. 23

[39] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proc. of the '93 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 146–157, 1993. 71

[40] P. Greenhalgh. Big.little processing with arm cortex-a15 & cortex-a7: Improving energy efficiency in high-performance mobile platforms, Sept. 2011. http://www.arm.com/files/downloads/big.LITTLE_Final.pdf. 45, 83

[41] B. Hackett. *Type Safety in the Linux Kernel*. PhD thesis, Stanford University, Apr. 2011. 41

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Conference Record of the 31st Annual ACM Symposium on Principles of Programming Languages*, pages 232–244, 2004. 40

[43] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(1):33–38, 2008. 83

[44] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–126, 2001. 90, 111

[45] Z. Hu, A. Buyuktosunoglu, V. Srinivisan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proc. of the 2004 International Symposium on Low Power Electronics and Design*, pages 32–37, 2004. 52

[46] IBM. *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*, 1999. 89

[47] Intel Corporation. *Intel Itanium 2 Processor Reference Manual: For Software Development and Optimization*, May 2004. 89

[48] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, volume 3B: System Programming Guide, Part 2*, Nov. 2006. 89, 110

[49] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 347–358, Dec. 2006. 83

[50] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient sat-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3):256–274, 2008. 40

[51] K. Jeong, A. B. Kahng, S. Kang, T. S. Rosing, and R. Strong. MAPG: Memory access power gating. In *Proc. of the 2012 Design, Automation and Test in Europe*, pages 1054–1059, 2012. 53

[52] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4), 2009. 10, 39

[53] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011. 9, 10, 42

[54] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 221–246, 2012. 42

[55] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt. Bottleneck identification and scheduling in multithreaded applications. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–234, 2012. 83, 85

[56] P. G. Joisha, R. S. Schreiber, P. Banerjee, H.-J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, 2011. 10, 42

[57] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, pages 81–92, Dec. 2003. 45, 83

[58] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, page 64, 2004. 45, 83

[59] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization*, pages 75–86, 2004. 32

[60] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power management of datacenter workloads using per-core power gating. *Computer Architecture Letters*, 8(2):48–51, 2009. 46, 52, 69, 81, 82

[61] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. In *Proc. of the 9th International Symposium on High-Performance Computer Architecture*, pages 113–122, 2003. 46

[62] J. Li, J. F. Martinez, and M. C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proc. of the 10th International Symposium on High-Performance Computer Architecture*, page 14, 2004. 85

[63] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. The McPAT framework for multicore and manycore architectures: Simultaneously modeling power, area, and timing. *ACM Transactions on Architecture and Code Optimization*, 10(1), 2013. 74

[64] C. Liu, A. Sivasubramaniam, M. Kademir, and M. J. Irwin. Exploiting barriers to optimize power consumption of cmps. In *Proc. of the 19th Int'l Parallel and Distributed Processing Symposium*, page 5a, 2005. 85

[65] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008. 4, 5

[66] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006. 5, 9, 41

[67] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of the '05 Conference on Programming Language Design and Implementation*, pages 190–200, 2005. 41, 92, 111

[68] A. Lukefahr, S. Padmanabha, R. Das, F. Sleiman, R. Dreslinski, T. Wenisch, and S. Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proc. of the 45th Annual International Symposium on Microarchitecture*, 2012. 45, 84

[69] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: Effective sampling for lightweight data-race detection. In *Proc. of the '09 Conference on Programming Language Design and Implementation*, pages 134–143, 2009. 41

[70] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pages 136–147, 1999. 89

[71] T. Miller, R. Thomas, and R. Teodorescu. Mitigating the effects of process variation in ultra-low voltage chip multiprocessors using dual supply voltages and half-speed units. *Computer Architecture Letters*, 11(2):45–48, 2012. 83

[72] T. N. Miller, X. Pan, R. Thomas, N. Sedaghti, and R. Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *Proc. of the 18th International Symposium on High-Performance Computer Architecture*, pages 1–12, 2012. 48, 51, 52, 77, 83, 85

[73] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 198–208, 2007. 91, 112

[74] K. G. Murty. *Linear Programming*. Wiley, 1983. 60

[75] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 446–455, 2007. 39

[76] A. Muzahid, N. Otsuki, and J. Torrellas. Atomtracker: A comprehensive approach to atomic region inference and violation detection. In *Proc. of the 43rd Annual International Symposium on Microarchitecture*, pages 287–297, 2010. 41

[77] MySQL: The World's Most Popular Open Source Database, 2012. http://www.mysql.com. 32

[78] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the '07 Conference on Programming Language Design and Implementation*, pages 89–100, 2007. 41, 92, 111

[79] OpenLDAP: Community Developed LDAP Software, 2012. http://www.openldap.org. 32

[80] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Publications, 1998. 57

[81] E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *Proc. of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 244, 2003. 71

[82] M. Prvulovic and J. Torrellas. Reenact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, 2003. 41

[83] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, 30(4):65–79, July 2010. 88, 89, 110

[84] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM TOCS*, 15(4):391–411, Nov. 1997. 9, 10

[85] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Conference Record of the 25th Annual ACM Symposium on Principles of Programming Languages*, pages 38–48, 1998. 40

[86] A. Sembrant, D. Black-Schaffer, and E. Hagersten. Phase guided profiling for fast cache modeling. In *Proc. of the 2012 International Symposium on Code Generation and Optimization*, pages 175–185, 2012. 112

[87] J. Sevcík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011. 10

[88] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Principles, 9th Edition*. John Wiley and Sons, Inc, Indianapolis, IN, 2012. 4

[89] R. Singhal. Inside intel next generation nehalem microarchitecture, 2008. http://software.intel.com/file/18976. 52, 74

[90] B. Sprunt. Performance monitoring hardware will always be a low priority, second class feature in processor design until.., Feb. 2005. In Workshop on Hardware Performance Monitors in conjunction with HPCA-11. 89

[91] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 253–264, 2009. 83, 85

[92] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 363–374, June 2008. 45, 83

[93] O. Traub, S. Schechter, and M. D. Smith. Ephemeral instrumentation for lightweight program profiling, 2000. 112

[94] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *Proc. of the 2005 International Symposium on Code Generation and Optimization*, pages 217–228, 2005. 89

[95] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, 1994. 63

[96] D. W. Wall. Predicting program behavior using real or estimated profiles. In *Proc. of the '91 Conference on Programming Language Design and Implementation*, pages 59–70, 1991. 108

[97] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, pages 209–220, 2007. 91, 112

[98] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 281–294, 2008. 9, 10, 12, 16, 40

[99] Y. Wang, H. Liao, S. Reveliotis, T. Kelly, S. Mahlke, and S. Lafortune. Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In *Proc. of the 48th IEEE Conference on Decision and Control*, pages 4971–4976, 2009. 15

[100] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, 1995. 5

[101] A. Youssef, M. Anis, and M. Elmasry. Dynmaic standby leakage prediction for leakage tolerant microprocessor functional units. In *Proc. of the 39th Annual International Symposium on Microarchitecture*, pages 371–384, 2006. 52

[102] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010. 41

[103] Q. Zhao, I. Cutcutache, and W.-F. Wong. Pipa: Pipelined profiling and analysis on multi-core systems. In *Proc. of the 2008 International Symposium on Code Generation and Optimization*, pages 185–194, 2008. 93, 113